The Pragmatic Programmers

TURING 图灵程序设计丛书

Async JavaScript

Build More Responsive Apps with Less Code

JavaScript异步编程

设计快速响应的网络应用

[美] Trevor Burnham 著

许青松 译



人民邮电出版社
POSTS & TELECOM PRESS

“别磨蹭了，快冲向购书网站，马上订购这本书吧！不管是浏览器端的JavaScript程序员，还是基于Node的JavaScript写手，都应该好好读读这本书。它不但为我们揭开了JavaScript事件模型的神秘面纱，而且具体指导了如何编写高效、高可读性的异步代码。强烈推荐你读一读！”

——Reginald Braithwaite，*CoffeeScript Ristretto* 一书作者

“我平时桌面上只留两本JavaScript方面的参考书，一本是犀牛书，一本是*Good Parts*。而这本书会是第三本。”

——Lon Ingram，Waterfall公司首席前端开发者

“对JavaScript的狂热促使Trevor奉献出了这本开卷有益的大作。过去一段时间，JavaScript的使用情景和复杂度与日俱增，服务器端运行时环境有增无减，客户端应用也在不断增长。此时，要使代码库更具可维护性、更优雅，异步模式就成为一种基本的选择。本书献上了关于异步编程模式的大量宝贵技巧、技术和指示，让我们几乎不费吹灰之力就能得偿夙愿。干得漂亮，Trevor！”

——Christophe Porteneuve，Delicious Insights公司CTO

JavaScript是个单线程的编程语言，你如何应对多媒体、多任务、多核的世界？经验丰富的JavaScript程序员也难免被网络中错综复杂的回调弄得灰头土脸。那么，你绝对应该看看这本《JavaScript异步编程》。

本书从最基本也是最重要的JavaScript事件模型开始，生动地复盘了各种异步应用情景，逐一呈现了目前在用的各种异步设计模式和异步编程类库，从PubSub到Promise对象，从异步工作流控制类库到worker多线程技术，直到浏览器端脚本的异步加载技术。本书叙述流畅，从问题引入，到初步解决，再到用例延伸、进阶方案，一路抽丝剥茧，层层推进，精彩纷呈。一册在手，定能让你自信地应对大型Web应用程序的复杂性，交付快速响应的JavaScript代码！

The Pragmatic Programmers

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-31657-8

9 787115 316578 >

ISBN 978-7-115-31657-8
定价：32.00元

TURING 图灵程序设计丛书

Async JavaScript

Build More Responsive Apps
with Less Code

JavaScript异步编程

设计快速响应的网络应用

[美] Trevor Burnham 著

许青松 译



人民邮电出版社

北 京

图书在版编目（C I P）数据

JavaScript异步编程 ： 设计快速响应的网络应用 /
(美) 伯纳姆 (Burnham,T.) 著 ; 许青松译. -- 北京 :
人民邮电出版社, 2013.5
（图灵程序设计丛书）
书名原文: Async JavaScript: Build More
Responsive Apps with Less Code
ISBN 978-7-115-31657-8

Ⅰ. ①J… Ⅱ. ①伯… ②许… Ⅲ. ①JAVA语言—程序
设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第079982号

内 容 提 要

本书讲述基本的异步处理技巧，包括 PubSub、事件模式、Promises 等，通过这些技巧，可以更好地应对大型 Web 应用程序的复杂性，交付快速响应的代码。理解了 Javascript 的异步模式可以让读者写出结构更合理、性能更出色、维护更方便的 Javascript 程序。

本书适合 JavaScript 开发人员阅读。

图灵程序设计丛书
JavaScript异步编程：设计快速响应的网络应用

◆ 著 [美] Trevor Burnham
译 许青松
责任编辑 岳新欣
执行编辑 李 瑛
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：880×1230 1/32
印张：4.375
字数：98千字 2013年5月第1版
印数：1－3 500册 2013年5月北京第1次印刷
著作权合同登记号 图字：01-2013-3130号
ISBN 978-7-115-31657-8

定价：32.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

本书献给史蒂夫·乔布斯和被他激励的一代企业家。

本书赞誉

本书谈论了目前 JavaScript 开发中最关键的主题之一：如何搞定并发性和并发任务，而不是被它们搞得抓狂！这可是目前我见过的第一本专注于该主题的著作。为了不被并发任务搞得抓狂，好好看看这本书吧。

——Peter Cooper，*JavaScript Weekly* 编辑

Trevor 简明扼要地阐释了如何编写异步的 JavaScript，并且不偏不倚地展示了浏览器端示例和服务器端示例。本书既是指南又是综述，十分引人入胜，任何打算打怪升级的 JavaScript 开发者都必须读一读。

——Wynn Netherland，Changelog 联合创办人

本书是 JavaScript 异步王国的终极指南。任何人只要想构建成熟的、良构的、高效的 JavaScript 应用，就必然要了解本书谈及的概念和工具。

——Julien Biezemans，Ruby/JavaScript 开发人员，
Cucumber.js 创建者

致　　谢

我对 JavaScript 算不上一见钟情，可如今她却是我最爱的两种编程语言之一。另外一种？当然是 JavaScript 的小妹——CoffeeScript 啦。我是如何放下忐忑之心，全心全意爱上 JavaScript 的呢？这个故事和成千上万名程序员的爱情故事没什么两样。我要感谢那些一开始就严肃对待 JavaScript 的人，正因为他们，才使得这门语言享有今天这样丰富的开发生态系统。感谢 John Resig 创建了浏览器端事实上的标准库 jQuery，感谢 Jeremy Ashkenas 造就了 CoffeeScript 和丰富但仍极尽简洁的 Backbone.js 框架，感谢 Ryan Dahl 赋予 JavaScript 一个健壮的服务器环境，还要感谢其他所有程序员通过自己的工作证明了 JavaScript 是一流的语言。

当然，如果只知道和 JavaScript 谈情说爱，我并不能写成这本书。我要感谢 Pragmatic Bookshelf 团队帮我彻底改造了那份匆匆写就的手稿，使它达到了 PragProg 闻名遐迩的优质标准。特别要感谢 Susannah Pfalzer 总编以及 Dave Thomas 和 Andy Hunt 这两位负责人，最最要感谢的是我的编辑 Jackie Carter。你们的理解和激励是我宝贵的财富。

此外还要感谢本书的技术审阅人：Julien Biezemans、Christophe

Porteneuve、Michael Ficarra、Travis Swicegood 和 Lon Ingram。特别要感谢 Karl Stolley，除了多次审阅手稿，还给予我很多其他方面的帮助。还要感谢 Stan Angeloff 和 Roly Fentanes 审阅了最初的手稿。书中遗留的任何错误都由我自己承担责任。

最后，感谢我供职的公司 HubSpot 一直支持我写完这本书。作为一位自由撰稿人，在多年流浪漂泊之后，我终于找到了归宿。

Trevor Burnham（trevorburnham@gmail.com）

2012 年 11 月

前　言

JavaScript 最初设计时是为了强化 Netscape 2.0 浏览器的网页表现力，现在却成为了多媒体、多任务、多内核网络世界中的一种单线程语言。不过从 1995 年算起，JavaScript 也不算是挣扎求存，倒可以算作茁壮成长。浏览器舞台上走马灯似地涌现出一个个潜在对手，你方唱罢我登场，随便举几个例子——Flash、Silverlight、Java 小应用，等等。

与此同时，出现了一个叫做 Ryan Dahl 的程序员。他想为事件驱动型的服务器建立一个新的框架，于是深入钻研计算机科学，苦苦寻找一种动态的、单线程的语言，最终却发现答案就在眼前。于是，Node.js 就此诞生，JavaScript 成为了服务器世界可以倚重的力量之一。

怎么会这样呢？2001 年，Paul Graham 在其文章“The Other Road Ahead”[①]中写下了这样的话：

---

① 此文章修订后的版本可见于 http://paulgraham.com/road.html。最初的脚注可见于 *Hackers & Painters* 一书，该书中文版《黑客和画家》已由人民邮电出版社出版。
——译者注

> 如果我是你，我甚至都不会碰 JavaScript……我在网上看到的大多数 JavaScript 都没有必要存在，而且其中很多代码都跑不通。

如今，Graham 是 Y Combinator 的首席合伙人，这家投资集团背后有 Dropbox、Heroku 以及数以百计正在运作的项目，几乎所有这些项目都使用了 JavaScript。正如 Graham 在修订后的文章中言道，"JavaScript 现在能用了"。

JavaScript 什么时候变成了一种体面的语言？有人说，转折点是 2004 年 Gmail 的问世。Gmail 向全世界证明了重量级的 Ajax 允许你在浏览器端运行一流的电子邮件客户端。还有人说，转折点是 2006 年 jQuery 的问世。jQuery 抽象出当时浏览器对手的 API，建立了事实上的标准。（截至 2011 年，最大的 1.7 万个网站中有 48%使用了 jQuery。[①]）

不管是什么原因，JavaScript 就坚守在这里。Apple 追随着 JavaScript 带来了 WebKit 和 Safari，Microsoft 追随着 JavaScript 带来了 Metro。甚至 Adobe 也捧 JavaScript 的场，其推出的一些工具开始生成 HTML5 而不是 Flash。JavaScript 一开始只是一种微不足道的浏览器特性，现在却理所当然地成为了世界上最重要的编程语言。

感谢网络浏览器的无所不在，JavaScript 比以往任何语言都更接近于兑现 Java 那句古老的承诺："一次编写，到处运行。"2007 年，Jeff Atwood 炮制出所谓的 Atwood 法则：

> 任何可以用 JavaScript 写成的应用最终都会用 JavaScript 写。[②]

---

① 参见 http://appendto.com/jquery-overtakes-flash。

② 参见 http://www.codinghorror.com/blog/2007/07/the-principle-of-least-power.html。

天堂里也有麻烦事儿

一般认为 JavaScript 是可以利用事件模型处理异步触发任务的单线程语言。如果只有两三个可能的事件，单线程语言编写的面向事件的代码要比多线程代码简单得多。这从概念上无懈可击，而且它不再需要用互斥量或信号量来封装数据以保证数据的线程安全性。但是，如果涌现出很多事件，同时要求数据的状态能够从一个事件传递到下一个事件，那么这种简单性常常要让位于令人望而却步的代码结构——金字塔厄运再次重现。

```
step1(function(result1) {
  step2(function(result2) {
    step3(function(result3) {
      // and so on...
    });
  });
});
```

“我喜欢异步编程，但我没法编出这样的代码。”这是 Node.js 谷歌小组中一位开发人员吐的苦水。[①]但问题并不在于语言，而在于程序员使用语言的方式。如何优雅地应对复杂的事件集，这仍然属于 JavaScript 有待解决的前沿领域。

那么，让我们继续向前冲吧！让我们证明给全世界看，即使是最复杂的问题也可以用整洁的、可维护的 JavaScript 代码来解决！

---

① 参见 https://groups.google.com/forum/#!topic/nodejs/wzSUdkPICWg。

本书读者

本书是写给中级 JavaScript 写手的。你应该了解变量的作用域是怎么回事，诸如 `typeof`、`arguments`、`this` 之类的关键字也不会让你如坠云雾。重点中的重点也许是你要知道

```
func(function(arg) { return next(arg); });
```

只是一种啰里啰嗦、毫无必要的写法，多数情况下它可以简写成：

```
func(next);
```

（请参阅 Reg Braithwaite 的文章“Captain Obvious on JavaScript”[①]，了解更多有关 JavaScript 用法的重要的小例子。）

目前你无需知道 JavaScript 如何调度异步事件,我们留待第 1 章再学习。

JavaScript的学习资源

随着 JavaScript 成为网络上的通用语（尚不论那些移动设备），涌现出了海量的参考书、课程和网站。下面是我推荐的一些学习资源。

- 如果你刚刚接触编程，请看看交互式的教程网站 Codecademy[②]。
- 如果你此前掌握了另一门编程语言，现在想将 JavaScript 作为浏览器的脚本语言，则请看看 CodeSchool 提供的交互式 jQuery 空中课堂[③]。

---

① 参见 https://github.com/raganwald/homoiconic/blob/master/2012/01/captain-obvious-onjavascript.md。

② 参见 http://www.codecademy.com/。

③ 参见 http://www.codeschool.com/。

- 如果你想要更正式一点的 JavaScript 语言介绍，请揣摩 Marijn Haverbeke 的 *Eloquent JavaScript*[①]一书 。
- 如果你只是 JavaScript 的初学者，想按部就班提高，避免掉入常见的陷阱，请花点时间看看 *JavaScript Garden*[②]。

如何求助

“这里应该用 `typeof` 还是 `instanceof` 呢？”对这样的问题举棋不定时，请绕开过气的 W3Schools 网站（遗憾的是，谷歌搜索用户很容易喜欢上它），直接访问 MDN（Mozilla Developer Network，Mozilla 开发人员网络）网站。[③]

Mozilla 基金会（你可能听过该基金会推出的 Firefox 浏览器）由 JavaScript 之父 Brendan Eich 主持。该基金会知道自己要干什么。

如果你在 MDN 的网页上找不到答案，可以带着问题去 Stack Overflow 网站[④]。这个网站有一个特别有用的开发者社区，而且我敢打赌，任何打上 JavaScript 标签的纠结问题都会很快有人回应。

运行代码示例

本书稍特别的地方在于，我会同时讨论客户端（浏览器）代码和服务器端（Node.js）代码。这也体现了 JavaScript 独特的可移植性。那些核

① 参见 http://eloquentjavascript.net/。
② 参见 http://javascriptgarden.info/。
③ 参见 https://developer.mozilla.org/。
④ 参见 http://stackoverflow.com/。

心概念适用于所有的 JavaScript 环境，但某些示例只针对某一种 JavaScript 环境。

即使你对编写 Node 应用毫无兴趣，也希望你在本地运行一遍这些代码片段。具体说明请参见“在 Node.js 中运行代码”。

可以运行哪些代码示例

如果看到的代码片段带有文件名，意味着这是一段独立的代码，无需更改即可运行。这里有一个例子：

Preface/stringConstructor.js

```
console.log('str'.constructor.name);
```

上下文环境会表明该代码可运行于浏览器端、Node.js 端，还是两者均可。

如果代码片段未带有文件名，则意味着这不是独立的代码，可能是大型代码例子的组成部分，或者是一段假想代码。例子如下。

```
var tenSeconds = 10 * 1e3;
setTimeout(launchSatellite, tenSeconds);
```

这些代码示例是用来阅读而不是用来运行的。

在Node.js中运行代码

Node 的安装和使用都非常简单：只要访问 http://nodejs.org/，单击 Download（下载），并运行 Windows 或 OS X 的安装程序（或者根据 *nix 源进行编译）。接下来，就可以从命令行运行 node 以开启一个

JavaScript REPL 会话（类似于 Ruby 的 irb 环境）。

```
$ node
> Math.pow(5, 6)
15625
```

将 JavaScript 文件名作为参数传给 node 命令，即可运行这个 JavaScript 文件。

```
$ echo "console.log(typeof NaN)" > foo.js
$ node foo.js
number
```

在浏览器中运行代码

目前的各个浏览器都提供了一个很小很好的 REPL 工具，支持在当前页面环境下运行 JavaScript 代码。不过要想玩转多行代码示例，最好还是脱机使用类似于 jsFiddle[①]这样的网络沙箱。

使用 jsFiddle，可以输入 JavaScript、HTML 及 CSS，然后单击 Run（运行）即可看到结果（或按 Ctrl+Enter）。（控制台输出将传递给开发人员的控制台。）也可以启用 jQuery 这样的框架，为此在左边栏选中它即可。还可以保存自己的活动，这会得到一个可分享的 URL。

本书的代码样式

JavaScript 没有官方的样式指南，但在项目中维持一致的编码样式又很重要。为此，本书采纳了以下非常常见的约定：

- 缩进为两个空格；

① 参见 http://jsfiddle.net/。

- 标识符遵循驼峰式大小写命名惯例；
- 每个表达式的末尾使用分号，但函数定义除外。

根据 Reg Braithwaite 的提议，我对函数调用链采用了特殊的缩进规则。这条缩进规则稍显复杂，基本而言是：当且仅当调用链中的两个函数调用返回同一个对象时，才会使用相同的缩进。因此，我会写出这样的示例：

```
$('#container > ul li.inactive')
.slideUp();
```

jQuery 的 `slideUp` 方法返回的对象就是调用自己的对象。于是，该函数调用不再缩进。与之相反：

```
var $paragraphClone = $('p:last')
  .clone();
```

这里的 `clone` 方法继续缩进，因为它返回了一个不同的对象。

这种约定的好处在于，它明确了调用链中各个函数的返回值。下面给出一个更复杂的示例。

```
$('h1')
  .first()
  .addClass('first')
.end()
  .last()
  .addClass('last');
```

jQuery 的 `first` 方法及 `last` 方法分别筛选一个数据集而剩下其第一个及最后一个元素，而 `end` 方法撤消了 `last` 筛选方法。于是，`end` 不再缩进，因为它返回的值等同于`$('h1')` 的值。（`last` 的缩进水平可以等同于 `first`，因为这时已重置了调用链。）

在进行函数式编程时，这种缩进方式特别有用，在第 4 章中将会看到这一点。

```
[1, 2, 3, 4, 5]
  .filter(function(int) { return int % 2 === 1; })
    .forEach(function(odd) { console.log(odd); })
```

关于altJS的只言片语

有很多语言能编译成 JavaScript，这可以简化代码的编写。（在 http://altjs.org 网站上能找到一个相当全面的语言清单。）本书不谈这些。本书谈论的是，即便不使用预编译器，也能编写出最棒的 JavaScript 代码。我对 altJS 没有任何成见（请参阅下一节），但我相信重点始终是如何理解底层的语言。

某些 altJS 语言专门致力于“驯化”异步调用，即可以用更偏向于同步的编码风格来编写异步调用。附录 1 会介绍这些语言的基本情况。

CoffeeScript

我喜爱 CoffeeScript，这不是什么秘密。CoffeeScript 是一种编译至 JavaScript 的优美表述性语言。我在 HubSpot 的日常工作中大量使用这种语言。在一些诸如 Railsconf、Øredev 之类的讨论会上，我也谈到了 CoffeScript。我的第一本书，*CoffeeScript: Accelerated JavaScript Develop- ment*，就是以 CoffeeScript 为主题的。[①]

① 参见 http://pragprog.com/book/tbcoffee/coffeescript。中文版《深入浅出 CoffeeScript》已由人民邮电出版社出版。——译者注

不过在开始撰写本书时我决定，用 CoffeeScript 来写只会得不偿失，徒然糟践了 CoffeeScript 的魅力。总的来说，CoffeeScript 写手对 JavaScript 的理解非常深刻，而诸如 `square = (x) => x * x` 之类的代码对于 JavaScript 纯化论者来说不啻于远古象形文字。[1]

所以，如果你是一位 CoffeeScript 程序员，我要为那些花括号向你致歉。至于其他人，请相信我，你从本书中学到的经验可以通行于任何 altJS 语言。

本书的相关资源

通过 The Pragmatic Bookshelf 网站上的本书页面（http://pragprog.com/book/tbajs/async-javascript），你可以下载本书中用到的示例代码，也可以获取最新的信息，还可以在一个热情友好的论坛里询问一些与本书有关的问题。

对于更常见的 JavaScript 相关问题，我（再次）衷心推荐 Stack Overflow 网站[2]。我跟该网站不存在什么瓜田李下之嫌，但我确实是它的热心拥趸，我在该网站的声望值已经骄傲地冲上 23 000 分大关（而且还继续累积着）。在这里，条清缕细、格式正规的问题几乎总能得到及时回应。

最后，如果你希望直接联系到我，可发邮件至 trevorburnham@gmail.com，或关注我的推特账号：@trevorburnham。我一直恭候着读者们的反馈。

闲言少絮。让我们异步起来吧！

---

① 好吧，也许不会一直这样，参见 http://wiki.ecmascript.org/doku.php?id=harmony:arrow_function_syntax。

② 参见 http://stackoverflow.com/。

目　录

第 1 章

深入理解 JavaScript 事件

事件！事件到底是怎么工作的？JavaScript 出现了多久，对 JavaScript 异步事件模型就迷惘了多久。迷惘导致 bug，bug 导致愤怒，然后尤达大师就会教我们如何如何……

不过本质上，从概念上看，JavaScript 事件模型既优雅又实用。一旦大家接受了这种语言的单线程设计，就会觉得 JavaScript 事件模型更像是一种功能，而不是一种局限。它意味着我们的代码是不可中断的，也意味着调度的事件会整整齐齐排好队，有条不紊地运行。

本章将介绍 JavaScript 的异步机制，并破除一些常见的误解。我们会看到 `setTimeout` 真正做了些什么。接着会讨论回调中抛出错误的处理。最后会奠定本书的主旨：为了清晰和可维护性，努力组织异步代码。

1.1 事件的调度

如果想让 JavaScript 中的某段代码将来再运行，可以将它放在回调中。回调就是一种普通函数，只不过它是传给像 `setTimeout` 这样的函数，或者绑定为像 `document.onready` 这样的属性。运行回调时，

我们称已触发某事件（譬如延时结束或页面加载完毕）。

当然，可怕的总是那些细节，哪怕是像 `setTimeout` 这样看起来很简单的东西。对 `setTimeout` 的描述通常像这样：

> 给定一个回调及 *n* 毫秒的延迟，`setTimeout` 就会在 *n* 毫秒后运行该回调。

但是，正如我们将在这一节乃至这一章里看到的，以上描述存在严重缺陷。大多数情况下，该描述只能算接近正确，而在其他情况下则完全是谬误。要想真正理解 `setTimeout`，必须先大体理解 JavaScript 事件模型。

1.1.1　现在还是将来运行

在探究 `setTimeout` 之前，先来看一个简单的例子。该情形常常会迷惑 JavaScript 新手，特别是那些刚刚从 Java 和 Ruby 等多线程语言迁移过来的新手。

EventModel/loopWithTimeout.js

```
for (var i = 1; i <= 3; i++) {
  setTimeout(function(){ console.log(i); }, 0);
};
```

```
4
4
4
```

大多数刚接触 JavaScript 语言的人都会认为以上循环会输出 `1，2，3`，或者重复输出这 3 个数字，因为这里的 3 次延时都抢着要第一个触发（每次暂停都调度为 0 毫秒后到时）。

要理解为什么输出是 4，4，4，需要知道以下 3 件事。

- 这里只有一个名为 i 的变量，其作用域由声明语句 var i 定义（该声明语句在不经意间让 i 的作用域不是循环内部，而是扩散至蕴含循环的那个最内侧函数）。
- 循环结束后，i===4 一直递增，直到不再满足条件 i<=3 为止。
- JavaScript 事件处理器在线程空闲之前不会运行。

前两条还属于 JavaScript 101 的范畴，但第三个更像是一个“惊喜”。一开始使用 JavaScript 的时候，我也不太相信会这样。Java 令我担心自己的代码随时会被中断。上百万种潜在的边界情况让我焦虑万分，我一直在想：“要是在这两行代码之间发生了什么稀奇古怪的事，会怎么样呢？”

然后，终于有一天，我再也没有这样的担心了……

1.1.2 线程的阻塞

下面这段代码打破了我对 JavaScript 事件的成见。

EventModel/loopBlockingTimeout.js

```
var start = new Date;
setTimeout(function(){
  var end = new Date;
  console.log('Time elapsed:', end - start, 'ms');
}, 500);
while (new Date - start < 1000) {};
```

按照多线程的思维定势，我会预计 500 毫秒后计时函数就会运行。不过这要求中断欲持续整整一秒钟的循环。如果运行代码，会得到类似这样的结果：

```
《Time elapsed: 1002ms
```

大家得到的数字可能会稍有不同，这是因为 `setTimeout` 和 `setInterval` 一样，其计时精度要比我们的期望值差很多（请参阅 1.2.2 节）。不过，这个数字肯定至少是 1000，因为 `setTimeout` 回调在 `while` 循环结束运行之前不可能被触发。

那么，如果 `setTimeout` 没有使用另一个线程，那它到底在做什么呢？

1.1.3　队列

调用 `setTimeout` 的时候，会有一个延时事件排入队列。然后 `setTimeout` 调用之后的那行代码运行，接着是再下一行代码，直到再也没有任何代码。这时 JavaScript 虚拟机才会问："队列里都有谁啊？"

如果队列中至少有一个事件适合于"触发"（就像 1000 毫秒之前设定好的那个为期 500 毫秒的延时事件），则虚拟机会挑选一个事件，并调用此事件的处理器（譬如传给 `setTimeout` 的那个函数）。事件处理器返回后，我们又回到队列处。

输入事件的工作方式完全一样：用户单击一个已附加有单击事件处理器的 DOM（Document Object Model，文档对象模型）元素时，会有一个单击事件排入队列。但是，该单击事件处理器要等到当前所有正在运行的代码均已结束后（可能还要等其他此前已排队的事件也依次结束）才会执行。因此，使用 JavaScript 的那些网页一不小心就会变得毫无反应。

你可能听过事件循环这个术语，它是用于描述队列工作方式的。所谓

事件循环，就像代码从一个循环中不断取出而运行一样：

```
runYourScript();
while (atLeastOneEventIsQueued) {
  fireNextQueuedEvent();
};
```

这隐含着一个意思，即触发的每个事件都会位于堆栈轨迹的底部。关于这一点，1.4 节会进一步阐述。

事件的易调度性是 JavaScript 语言最大的特色之一。像 `setTimeout` 这样的异步函数只是简单地做延迟执行，而不是孵化新的线程。JavaScript 代码永远不会被中断，这是因为代码在运行期间只需要排队事件即可，而这些事件在代码运行结束之前不会被触发。

下一节将更细致地考查异步 JavaScript 代码的构造块。

1.2　异步函数的类型

每一种 JavaScript 环境都有自己的异步函数集。有些函数，如 `setTimeout` 和 `setInterval`，是各种 JavaScript 环境普遍都有的。另一些函数则专属于某些浏览器或某几种服务器端框架。JavaScript 环境提供的异步函数通常可以分为两大类：I/O 函数和计时函数。如果想在应用中定义复杂的异步行为，就要使用这两类异步函数作为基本的构造块。

1.2.1　异步的I/O函数

创造 Node.js，并不是为了人们能在服务器上运行 JavaScript，仅仅是因为 Ryan Dahl 想要一个建立在某高级语言之上的事件驱动型服务器

框架。JavaScript 碰巧就是适合干这个的语言。为什么？因为 JavaScript 语言可以完美地实现非阻塞式 I/O。

在其他语言中，一不小心就会“阻塞”应用（通常是运行循环）直到完成 I/O 请求为止。而在 JavaScript 中，这种阻塞方式几乎沦为无稽之谈。类似如下的循环将永远运行下去，不可能停下来。

```
var ajaxRequest = new XMLHttpRequest;
ajaxRequest.open('GET', url);
ajaxRequest.send(null);
while (ajaxRequest.readyState === XMLHttpRequest.UNSENT) {
  // readyState 在循环返回之前不会有更改。
};
```

相反，我们需要附加一个事件处理器，随即返回事件队列。

```
var ajaxRequest = new XMLHttpRequest;
ajaxRequest.open('GET', url);
ajaxRequest.send(null);
ajaxRequest.onreadystatechange = function() {
  // ...
};
```

就是这么回事。不论是在等待用户的按键行为，还是在等待远程服务器的批量数据，所需要做的就是定义一个回调，除非 JavaScript 环境提供的某个同步 I/O 函数已经替我们完成了阻塞。

在浏览器端，Ajax 方法有一个可设置为 `false` 的 `async` 选项（但永远、永远别这么做），这会挂起整个浏览器窗格直到收到应答为止。在 Node.js 中，同步的 API 方法在名称上会有明确的标示，譬如 `fs.readFileSync`。编写短小的脚本时，这些同步方法会很方便。但是，如果所编写的应用需要处理并行的多个请求或多项操作，则应该避免使用它们。可在今天，还有哪个应用不是这样的呢？

有些 I/O 函数既有同步效应，也有异步效应。举例来说，在现代浏览器中操纵 DOM 对象时，从脚本角度看，更改是即时生效的，但从视效角度看，在返回事件队列之前不会渲染这些 DOM 对象更改。这可以防止 DOM 对象被渲染成不一致的状态。关于这点，可访问 http://jsfiddle.net/ TrevorBurnham/SNBYV/，查看一个简单的演示。

console.log是异步的吗？

WebKit的`console.log`由于表现出异步行为而让很多开发者惊诧不已。在Chrome或Safari中，以下这段代码会在控制台记录`{foo:bar}`。

EventModel/log.js

```
var obj = {};
console.log(obj);
obj.foo = 'bar';
```

怎么会这样？WebKit的`console.log`并没有立即拍摄对象快照，相反，它只存储了一个指向对象的引用，然后在代码返回事件队列时才去拍摄快照。

Node的`console.log`是另一回事，它是严格同步的，因此同样的代码输出的却为`{}`。

JavaScript 采用了非阻塞式 I/O，这对新手来说是最大的一个障碍，但这同样也是该语言的核心优势之一。有了非阻塞式 I/O，就能自然而然地写出高效的基于事件的代码。

1.2.2 异步的计时函数

我们已经看到，异步函数非常适合用于 I/O 操作，但有些时候，我们仅仅是因为需要异步而想要异步性。换句话说，我们想让一个函数在

将来某个时刻再运行——这样的函数可能是为了作动画或模拟。基于时间的事件涉及两个著名的函数，即 setTimeout 与 setInterval。

遗憾的是，这两个著名的计时器函数都有自己的一些缺陷。正如我们在 1.1.2 节中看到的，其中有个缺陷是无法弥补的：当同一个 JavaScript 进程正运行着代码时，任何 JavaScript 计时函数都无法使其他代码运行起来。但是，即便容忍了这一局限性，setTimeout 及 setInterval 的不确定性也会令人犯怵。下面是一个示例。

EventModel/fireCount.js

```
var fireCount = 0;
var start = new Date;
var timer = setInterval(function() {
  if (new Date-start > 1000) {
    clearInterval(timer);
    console.log(fireCount);
    return;
  }
  fireCount++;
}, 0);
```

如果使用 setInterval 调度事件且延迟设定为 0 毫秒，则会尽可能频繁地运行此事件，对吗？那么，在运行于高速英特尔 i7 处理器之上的现代浏览器中，此事件的触发频率到底如何呢？

大约为 200 次/秒。这是 Chrome、Safari 和 Firefox 等浏览器的平均值。在 Node 环境下，此事件的触发频率大约能达到 1000 次/秒。（若使用 setTimeout 来调度事件，重复这些实验也会得到类似的结果。）作为对比，如果将 setInterval 替换成简单的 while 循环，则在 Chrome 中此事件的触发频率将达到 400 万次/秒，而在 Node 中会达到 500 万次/秒！

这是怎么回事？最后我们发现，`setTimeout` 和 `setInterval` 就是想设计成慢吞吞的！事实上，HTML 规范（这是所有主要浏览器都遵守的规范）推行的延时/时隔的最小值就是 4 毫秒！①

那么，如果需要更细粒度的计时，该怎么办呢？有些运行时环境提供了备选方案。

- 在 Node 中，`process.nextTick` 允许将事件调度成尽可能快地触发。对于笔者的系统，`process.nextTick` 事件的触发频率可以超过 10 万次/秒。
- 一些现代浏览器（含 IE9+）带有一个 `requestAnimationFrame` 函数。此函数有两个目标：一方面，它允许以 60+帧/秒的速度运行 JavaScript 动画；另一方面，它又避免后台选项卡运行这些动画，从而节约 CPU 周期。在最新版的 Chrome 浏览器中，甚至能实现亚毫秒级的精度。②

尽管这些计时函数是异步 JavaScript 混饭吃的家伙什儿，但永远不要忘记，`setTimeout` 和 `setInterval` 就是些不精确的计时工具。在 Node 中，如果只是想产生一个短时延迟，请使用 `process.nextTick`。在浏览器端，请尝试使用垫片技术（shim）③：在支持 `requestAnimationFrame` 的浏览器中，推荐使用 `requestAnimationFrame`；在不支持 `requestAnimationFrame` 的

① 参见 http://www.whatwg.org/specs/web-apps/current-work/multipage/timers.html#dom-windowtimers-settimeout。

② 参见 http://updates.html5rocks.com/2012/05/requestAnimationFrame-API-now-with-sub-millisecond-precision。

③ http://paulirish.com/2011/requestanimationframe-for-smart-animating/。垫片技术可简述为，它负责将一个新的 API 引入到一个旧的环境中，且仅仅依靠旧环境中已有的手段来实现。——译者注

浏览器中，则退而使用 `setTimeout`。

到这里，关于 JavaScript 基本异步函数的简要概览就结束了。但怎样才能知道一个函数到底何时异步呢？下一节中，我们在亲自编写异步函数的同时再思考这个问题。

1.3　异步函数的编写

JavaScript 中的每个异步函数都构建在其他某个或某些异步函数之上。凡是异步函数，从上到下（一直到原生代码）都是异步的！

反之亦然：任何函数只要使用了异步的函数，就必须以异步的方式给出其操作结果。正如我们在 1.1.2 节学到的，JavaScript 并没有提供一种机制以阻止函数在其异步操作结束之前返回。事实上，除非函数返回，否则不会触发任何异步事件。

本节将考察异步函数设计的一些常见模式。我们将看到有些函数如反复无常的小人，非得等到特定时候才下决心成为异步的。不过，我们先来精确地定义异步函数。

1.3.1　何时称函数为异步的

*异步函数*这个术语有点名不副实：调用一个函数时，程序只在该函数返回之后才能继续。JavaScript 写手如果称一个函数为“异步的”，其意思是这个函数会导致将来再运行另一个函数，后者取自于事件队列（若后面这个函数是作为参数传递给前者的，则称其为*回调函数*，简称为*回调*）。于是，一个取用回调的异步函数永远都能通过以下测试。

```
var functionHasReturned = false;
asyncFunction(function() {
  console.assert(functionHasReturned);
});
functionHasReturned = true;
```

异步函数还涉及另一个术语，即非阻塞。非阻塞这个词强调了异步函数的高速度：异步 MySQL 数据库驱动程序做一个查询可能要花上一小时，但负责发送查询请求的那个函数却能以微秒级速度返回。这对于那些需要快速处理海量请求的网站服务器来说，绝对是个福音。

通常，那些取用回调的函数都会将其作为自己的最后一个参数。（可惜的是，老资格的 `setTimeout` 和 `setInterval` 都是这一约定的特例。）不过，有些异步函数也会间接取用回调，它们会返回 Promise 对象或使用 PubSub 模式。本书稍后就会介绍这些异步设计模式。

遗憾的是，要想确认某个函数异步与否，唯一的方法就是审查其源代码。有些同步函数却拥有看起来像是异步的 API，这或者是因为它们将来可能会变成异步的，又或者是因为回调这种形式能方便地返回多个参数。一旦存疑，请别指望函数就是异步的。

1.3.2 间或异步的函数

有些函数某些时候是异步的，但其他时候却不然。举个例子，jQuery 的同名函数（通常记作`$`）可用于延迟函数直至 DOM 已经结束加载。但是，若 DOM 早已结束了加载，则不存在任何延迟，`$`的回调将会立即触发。

不注意的话，这种行为的不可预知性会带来很多麻烦。我曾经看到也犯过这样一个错误，即假定`$`会在已加载本页面其他脚本之后再运行

一个函数。

```
// application.js
$(function() {
  utils.log('Ready');
});

// utils.js
window.utils = {
  log: function() {
    if (window.console) console.log.apply(console, arguments);
  }
};

<script src ="application.js"></script>
<script src ="util.js"></script>
```

这段代码运行得很好，但前提是浏览器并未从缓存中加载页面（这会导致 DOM 早在脚本运行之前就已加载就绪）。如果出现这种情况，传递给$的回调就会在设置 `utils.log` 之前运行，从而导致一个错误。（为了避免这种情况，应该采用一种更现代的管理客户端依赖性的方法。请参阅第 6 章。）

下面来看另一个例子。

1.3.3　缓存型异步函数

间或异步的函数有一个常见变种是可缓存结果的异步请求类函数。举例来说，假设正在编写一个基于浏览器的计算器，它使用了网页 Worker 对象以单独开一个线程来进行计算。（第 5 章将介绍网页 Worker 对象的 API。）主脚本看起来像这样：[①]

① 访问 http://webworkersandbox.com/5009efc12245588e410002cf，可以看到这个例子的一个可运行版本。

```
var calculationCache = {},
    calculationCallbacks = {},
    mathWorker = new Worker('calculator.js');

mathWorker.addEventListener('message', function(e) {
  var message = e.data;
  calculationCache[message.formula] = message.result;
  calculationCallbacks[message.formula](message.result);
});

function runCalculation(formula, callback) {
  if (formula in calculationCache) {
    return callback(calculationCache[formula]);
  };
  if (formula in calculationCallbacks) {
    return setTimeout(function() {
      runCalculation(formula, callback);
    }, 0);
  };
  mathWorker.postMessage(formula);
  calculationCallbacks[formula] = callback;
}
```

在这里，当结果已经缓存时，`runCalculation` 函数是同步的，否则就是异步的。存在 3 种可能的情景。

- 公式已经计算完成，于是结果位于 `calculationCache` 中。这种情况下，`runCalculation` 是同步的。
- 公式已经发送给 Worker 对象，但尚未收到结果。这种情况下，`runCalculation` 设定了一个延时以便再次调用自身；重复这一过程直到结果位于 `calculationCache` 中为止。
- 公式尚未发送给 Worker 对象。这种情况下，将会从 Worker 对象的 `'message'`事件监听器激活回调。

请注意，在第 2 种和第 3 种情景中，我们按照两种不同的方式来等待任务的完成。这个例子写成这样，就是为了演示依据哪几种常见方式来等待某些东西发生改变（如缓存型计算公式的值）。是不是应该倾向于其中某种方式呢？我们接着往下看。

1.3.4　异步递归与回调存储

在 runCalculation 函数中，为了等待 Worker 对象完成自己的工作，或者通过延时而重复相同的函数调用（即*异步递归*），或者简单地存储回调结果。

哪种方式更好呢？乍一看，只使用异步递归是最简单的，因为这里不再需要 calculationCallbacks 对象。出于这个目的，JavaScript 新手常常会使用 setTimeout，因为它很像线程型语言的风格。此程序的 Java 版本可能会有这样一个循环：

```
while (!calculationCache.get(formula)) {
  Thread.sleep(0);
};
```

但是，延时并不是免费的午餐。大量延时的话，会造成巨大的计算荷载。异步递归有一点很可怕，即在等待任务完成期间，可触发之延时的次数是不受限的！此外，异步递归还毫无必要地复杂化了应用程序的事件结构。基于这些原因，应将异步递归视作一种“反模式”的方式。

在这个计算器例子中，为了避免异步递归，可以为每个公式存储一个回调数组。

```
var calculationCache = {},
    calculationCallbacks = {},
    mathWorker = new Worker('calculator.js');
mathWorker.addEventListener('message', function(e) {
  var message = e.data;
  calculationCache[message.formula] = message.result;
  calculationCallbacks[message.formula]
  .forEach(function(callback) {
    callback(message.result);
  });
});

function runCalculation(formula, callback) {
  if (formula in calculationCache) {
    return callback(calculationCache[formula]);
  };
  if (formula in calculationCallbacks) {
    return calculationCallbacks[formula].push(callback);
  };
  mathWorker.postMessage(formula);
  calculationCallbacks[formula] = [callback];
}
```

没有了延时，我们的代码要直观得多，也高效得多。

总的来说，请避免异步递归。仅当所采用的库提供了异步功能但没有提供任何形式的回调机制时，异步递归才有必要。如果真的遇到这种情况，要做的第一件事应该是为该库写一个补丁。或者，干脆找一个更好的库。

1.3.5 返值与回调的混搭

在以上两种 `runCalculation` 实现中，有时会用到返值技术。这是出于简洁的目的而随意作出的选择。下面这行代码

```
return callback(calculationCache[formula]);
```

很容易即可改写成

```
callback(calculationCache[formula]);
return;
```

这是因为并没有打算使用这个返值。这是 JavaScript 的一种普遍做法，而且通常无害。

不过，有些函数既返回有用的值，又要取用回调。这类情况下，切记回调有可能被同步调用（返值之前），也有可能被异步调用（返值之后）。

永远不要定义一个潜在同步而返值却有可能用于回调的函数。举个例子，下面这个负责打开 WebSocket[①]连接以连至给定服务器的函数（使用缓存技术以确保每个服务器只有一个连接）就违反了上述规则。

```
var webSocketCache = {};
function openWebSocket(serverAddress, callback) {
  var socket;

  if (serverAddress in webSocketCache) {
    socket = webSocketCache[serverAddress];

    if (socket.readyState === WebSocket.OPEN) {
      callback();
    } else {
      socket.onopen = _.compose(callback, socket.onopen);
    };
  } else {
    socket = new WebSocket(serverAddress);
    webSocketCache[serverAddress] = socket;
```

① 参见 https://developer.mozilla.org/en/WebSockets/。

```
    socket.onopen = callback;
  };
  return socket;
};
```

（这段代码依赖于 Underscore.js 库。`_.compose` 定义的这个新函数既运行了 `callback`，又运行了初始的 `socket.onopen` 回调。[①]）

这段代码的问题在于，如果套接字已经缓存且打开，则会在函数返值之前就运行回调，这会使以下代码崩溃。

```
var socket = openWebSocket(url, function() {
  socket.send('Hello, server!');
});
```

怎么解决呢？将回调封装在 `setTimeout` 中即可。

```
if (socket.readyState === WebSocket.OPEN) {
  setTimeout(callback, 0);
} else {
  // ...
}
```

这里使用延时会让人感觉是在东拼西凑，但这总比 API 自相矛盾要好得多。

在本节中，我们看到了一些编写异步函数的最佳实践。请勿依赖那些看似始终异步的函数，除非已经阅读其源代码。请避免使用计时器方法来等待某个会变化的东西。如果同一个函数既返值又运行回调，则请确保回调在返值之后才运行。

一次消化这些信息确实太多了一点，不过，编写好的异步函数确实是

① 参见 http://documentcloud.github.com/underscore/#compose。

写出优秀 JavaScript 代码的关键所在。

1.4　异步错误的处理

像很多时髦的语言一样，JavaScript 也允许抛出异常，随后再用一个 `try/catch` 语句块捕获。如果抛出的异常未被捕获，大多数 JavaScript 环境都会提供一个有用的堆栈轨迹。举个例子，下面这段代码由于`'{'`为无效 JSON 对象而抛出异常。

EventModel/stackTrace.js

```
function JSONToObject(jsonStr) {
  return JSON.parse(jsonStr);
}
var obj = JSONToObject('{');
```

```
《SyntaxError: Unexpected end of input
    at Object.parse (native)
    at JSONToObject (/AsyncJS/stackTrace.js:2:15)
    at Object.<anonymous> (/AsyncJS/stackTrace.js:4:11)
```

堆栈轨迹不仅告诉我们哪里抛出了错误，而且说明了最初出错的地方：第 4 行代码。遗憾的是，自顶向下地跟踪异步错误起源并不都这么直截了当。在本节中，我们会看到为什么 `throw` 很少用作回调内错误处理的正确工具，还会了解如何设计异步 API 以绕开这一局限。

1.4.1　回调内抛出的错误

如果从异步回调中抛出错误，会发生什么事？让我们先来做个测试。

EventModel/nestedErrors.js

```
setTimeout(function A() {
  setTimeout(function B() {
```

```
    setTimeout(function C() {
      throw new Error('Something terrible has happened!');
    }, 0);
  }, 0);
}, 0);
```

上述应用的结果是一条极其简短的堆栈轨迹。

```
《Error: Something terrible has happened!
    at Timer.C (/AsyncJS/nestedErrors.js:4:13)
```

等等，A 和 B 发生了什么事？为什么它们没有出现在堆栈轨迹中？这是因为运行 C 的时候，A 和 B 并不在内存堆栈里。这 3 个函数都是从事件队列直接运行的。

基于同样的理由，利用 try/catch 语句块并不能捕获从异步回调中抛出的错误。下面进行演示。

EventModel/asyncTry.js

```
try {
  setTimeout(function() {
    throw new Error('Catch me if you can!');
  }, 0);
} catch (e) {
  console.error(e);
}
```

看到这里的问题了吗？这里的 try/catch 语句块只捕获 setTimeout 函数自身内部发生的那些错误。因为 setTimeout 异步地运行其回调，所以即使延时设置为 0，回调抛出的错误也会直接流向应用程序的未捕获异常处理器（请参阅 1.4.2 节）。

总的来说，取用异步回调的函数即使包装上 try/catch 语句块，也只是无用之举。（特例是，该异步函数确实是在同步地做某些事且容

易出错。例如，Node 的 `fs.watch(file,callback)`就是这样一个函数，它在目标文件不存在时会抛出一个错误。）正因为此，Node.js 中的回调几乎总是接受一个错误作为其首个参数，这样就允许回调自己来决定如何处理这个错误。举个例子，下面这个 Node 应用尝试异步地读取一个文件，还负责记录下任何错误（如“文件不存在”）。

EventModel/readFile.js

```
var fs = require('fs');
fs.readFile('fhgwgdz.txt', function(err, data) {
  if (err) {
    return console.error(err);
  };
  console.log(data.toString('utf8'));
});
```

客户端 JavaScript 库的一致性要稍微差些，不过最常见的模式是，针对成败这两种情形各规定一个单独的回调。jQuery 的 Ajax 方法就遵循了这个模式。

```
$.get('/data', {
  success: successHandler,
  failure: failureHandler
});
```

不管 API 形态像什么，始终要记住的是，只能在回调内部处理源于回调的异步错误。异步尤达大师会说：“做，或者不做，没有试试看一说。”

1.4.2　未捕获异常的处理

如果是从回调中抛出异常的，则由那个调用了回调的人负责捕获该异常。但如果异常从未被捕获，又会怎么样？这时，不同的 JavaScript

环境有着不同的游戏规则……

1. 在浏览器环境中

现代浏览器会在开发人员控制台显示那些未捕获的异常，接着返回事件队列。要想修改这种行为，可以给 `window.onerror` 附加一个处理器。如果 `windows.onerror` 处理器返回 `true`，则能阻止浏览器的默认错误处理行为。

```
window.onerror = function(err) {
  return true;  //彻底忽略所有错误
};
```

在成品应用中，会考虑某种 JavaScript 错误处理服务，譬如 Errorception[1]。Errorception 提供了一个现成的 `windows.onerror` 处理器，它向应用服务器报告所有未捕获的异常，接着应用服务器发送消息通知我们。

2. 在 Node.js 环境中

在 Node 环境中，`window.onerror` 的类似物就是 `process` 对象的 `uncaughtException` 事件。正常情况下，Node 应用会因未捕获的异常而立即退出。但只要至少还有一个 `uncaughtException` 事件处理器，Node 应用就会直接返回事件队列。

```
process.on('uncaughtException', function(err) {
  console.error(err);  //避免了关停的命运！
});
```

但是，自 Node 0.8.4 起，`uncaughtException` 事件就被废弃了。据

① 参见 http://errorception.com/。

其文档[①]所言，

> 对异常处理而言，uncaughtException 是一种非常粗暴的机制，它在将来可能会被放弃……
>
> 请勿使用 uncaughtException，而应使用 Domain 对象。

Domain 对象又是什么？你可能会这样问。Domain 对象是事件化对象（第 2 章会详细讨论），它将 throw 转化为'error'事件。下面是一个例子。

EventModel/domainThrow.js

```
var myDomain = require('domain').create();
myDomain.run(function() {
  setTimeout(function() {
    throw new Error('Listen to me!')
  }, 50);
});

myDomain.on('error', function(err) {
  console.log('Error ignored!');
});
```

源于延时事件的 throw 只是简单地触发了 Domain 对象的错误处理器。

```
❮Error ignored!
```

很奇妙，是不是？Domain 对象让 throw 语句生动了很多。遗憾的是，仅在 Node 0.8+环境中才能使用 Domain 对象；在我写作本书时，Domain 对象仍被视作试验性的特性。更多信息请参阅 Node 文档。[②]

不管在浏览器端还是服务器端，全局的异常处理器都应被视作最后一

① 参见 http://nodejs.org/docs/latest/api/process.html#process_event_uncaughtexception。
② 参见 http://nodejs.org/docs/latest/api/domain.html。

根救命稻草。请仅在调试时才使用它。

1.4.3　抛出还是不抛出

遇到错误时，最简单的解决方法就是抛出这个错误。在 Node 代码中，大家会经常看到类似这样的回调：

```
function(err) {
  if (err) throw err;
  // ...
}
```

在第 4 章中，我们会经常沿用这一做法。但是，在成品应用中，允许例行的异常及致命的错误像踢皮球一样踢给全局处理器，这是不可接受的。回调中的 `throw` 相当于 JavaScript 写手在说“现在我还不想考虑这个”。

如果抛出那些自己知道肯定会被捕获的异常呢？这种做法同样凶险万分。2011 年，Isaac Schlueter（npm 的开发者，在任的 Node 开发负责人）就主张 `try/catch` 是一种“反模式”的方式。[①]

> `try/catch` 只是包装着漂亮花括弧的 goto 语句。一旦跑去处理错误，就无法回到中断之处继续向下执行。更糟糕的是，通过 `throw` 语句的代码，完全不知道自己会跳到什么地方。返回错误码的时候，就相当于正在履行合约。抛出错误的时候，就好像在说，“我知道我正在和你说话，但我现在不想搭理你，我要先找你老板谈谈”，这太粗俗无礼了。如果不是什么紧急情况，请别这么做；如果确实是紧急情况，则应该直接崩溃掉。

① 参见 https://groups.google.com/forum/#!topic/nodejs/1ESsssIxrUU。

Schlueter 提倡完全将 `throw` 用作断言似的构造结构，作为一种挂起应用的方式——当应用在做完全没预料到的事时，即挂起应用。Node 社区主要遵循这一建议，尽管这种情况可能会随着 Domain 对象的出现而改变。

那么，关于异步错误的处理，目前的最佳实践是什么呢？我认为应该听从 Schlueter 的建议：如果想让整个应用停止工作，请勇往直前地大胆使用 `throw`。否则，请认真考虑一下应该如何处理错误。是想给用户显示一条出错消息吗？是想重试请求吗？还是想唱一曲“雏菊铃之歌”[①]？那就这么处理吧，只是请尽可能地靠近错误源头。

1.5 嵌套式回调的解嵌套

JavaScript 中最常见的反模式做法是，回调内部再嵌套回调。还记得前言里提到的金字塔厄运吗？我们先来看一个具体的例子，你也可能在 Node 服务器上看到过类似的代码。

```
function checkPassword(username, passwordGuess, callback) {
  var queryStr = 'SELECT * FROM user WHERE username = ?';
  db.query(queryStr, username, function (err, result) {
    if (err) throw err;
    hash(passwordGuess, function(passwordGuessHash) {
      callback(passwordGuessHash === result['password_hash']);
    });
  });
}
```

这里定义了一个异步函数 `checkPassword`，它触发了另一个异步函数 `db.query`，而后者又可能触发另外一个异步函数 `hash`。（在

① 1892 年著名的歌曲 *Daisy Bell*，这是第一首由电脑模拟人声唱出的歌曲。——译者注

阅读代码之前，无法确认这些函数是否真的异步，但这里的几个函数理应如此。）

这段代码有什么问题呢？目前为止，没有任何问题。它能用，而且简洁明了。但是，如果试图向其添加新特性，它就会变得毛里毛躁、险象环生，比如去处理那个数据库错误，而不是抛出错误（请参阅 1.4.3 节）、记录尝试访问数据库的次数、阻塞访问数据库，等等。

嵌套式回调诱惑我们通过添加更多代码来添加更多特性，而不是将这些特性实现为可管理、可重用的代码片段。checkPassword 有一种可以避免出现上述苗头的等价实现方式，如下：

```
function checkPassword(username, passwordGuess, callback) {
  var passwordHash;
  var queryStr = 'SELECT * FROM user WHERE username = ?';
  db.query(qyeryStr, username, queryCallback);

  function queryCallback(err, result) {
    if (err) throw err;
    passwordHash = result['password_hash'];
    hash(passwordGuess, hashCallback);
  }

  function hashCallback(passwordGuessHash) {
    callback(passwordHash === passwordGuessHash);
  }
}
```

这种写法更啰嗦一些，但读起来更清晰，也更容易扩展。由于这里赋予了异步结果（即 passwordHash）更宽广的作用域，所以获得了更大的灵活性。

按照惯例，请避免两层以上的函数嵌套。关键是找到一种在激活异步

调用之函数的外部存储异步结果的方式，这样回调本身就没有必要再嵌套了。

如果这样听起来有点诘聱难懂，请别担心。我们在后续几章中会看到大量的异步事件例子，那里的异步事件顺序运行且没有嵌套式事件处理器。

1.6　小结

本章阐释了 JavaScript 的单线程性为什么既是福利又是祸害。使用得当的话，它会使代码优美且没有那些多线程应用中泛滥成灾的可怕竞态条件。不过，这需要你形成正确的思维定势并掌握恰当的技术。

本书其余章节将介绍 JavaScript 中处理事件时用到的一些库和设计模式。我们考查的所有示例都可以运行于主流的浏览器或未经改动的 Node.js 环境。不过，编写 JavaScript 并不是产生 JavaScript 代码的唯一途径。关于其他一些有趣编辑器的概况，请参阅附录 A。

这里值得提一下，JavaScript 中存在一种多线程性：可以孵化出 Worker 进程。每个孵化出的进程都可以与其他进程交换数据，其限制等同于任何其他 I/O 进程。Worker 对象使得我们有可能利用多个内核，同时不会破坏 JavaScript 的游戏规则（代码不可能被中断；变量只有处于其作用域内部时才是可访问的）。关于 Worker 对象的更多内容，请参见第 5 章。

接下来两章将专门讨论两种基本的设计模式。PubSub 模式是一种将回调赋值给已命名事件的回调组织方式，而 Promise 对象是一种表示一次性事件的直观对象。

第 2 章

分布式事件

在上一章中，我们了解了 JavaScript 异步事件的工作方式。但在实践中到底应该怎样处理这些事件呢?

这个问题听起来好像很愚蠢。直接给应用程序关心的每个事件都附加一个处理器不就行了吗？然而，一旦单一的事件有着多重的后果，这种“一事一处理”的方式将迫使处理器规模急剧膨胀。

假设我们正在构建一个类似于 Google Docs 的网页版文字处理程序。每当用户按下一个键时，都要做很多事情：新键入的字符必须显示在屏幕上；插入点必须向后移动；这次键入动作必须推入本地的撤销动作历史记录中，且必须与服务器进行同步；拼写检查功能也必须运行起来；字数统计和页数统计也需要加以更新。用一个 `keypress` 处理器就想完成所有这些任务甚至更多任务，这显然会令人望而却步。

从纯机械论的角度看，每项因响应事件而执行的任务都确实必须由事件处理器发起。但是，从人类感性的角度出发，这个庞大的事件处理器通常最好能替换成更具延展性的、动态的构造——一种可以在运行时对其增减任务的构造。简而言之，我们希望使用分布式事件：事件的蝴蝶偶然扇动了下翅膀，整个应用到处都引发了反应。

在本章中，你将会学到如何使用 PubSub（Publish/Subscribe，意为“发布/订阅”）模式来分发事件。沿着这个思路，我们会看到 PubSub 模式的一些具体表现：Node 的 `EventEmitter` 对象、Backbone 的事件化模型和 jQuery 的自定义事件。在这些工具的帮助下，我们能解嵌套那些嵌套式回调，减少重复冗余，最终编写出易于理解的事件驱动型代码。

2.1　PubSub 模式

从 JavaScript 诞生之日起，浏览器就允许向 DOM 元素附加事件处理器，形如：

```
link.onclick = clickHandler;
```

啊哈，一目了然！只不过要提醒你一点：如果想向一个元素附加两个点击事件处理器，则必须自行用一个封装函数汇集这两个处理器。

```
link.onclick = function() {
  clickHandler1.apply(this, arguments);
  clickHandler2.apply(this, arguments);
};
```

这不仅冗长重复，而且也会制造出浮肿的、“全能的”处理器函数。正因为此，W3C 于 2000 年向 DOM 规范中添加了 `addEventListener` 方法，而 jQuery 将其抽象成 `bind` 方法。使用 `bind`，很容易对任何元素或元素集合发生的任何事件添加任意多的处理器，且完全不用担心这些处理器因摩肩接踵而出现踩踏事故。

```
$(link)
  .bind('click', clickHandler1)
```

```
  .bind('click', clickHandler2);
```

（在 jQuery1.7+中，优先使用新的 on 语法而不用 bind。[1]那里也提供了 click 方法，不过它只是 bind('click',...)的简写。但是，笔者倾向于一直使用 bind/on。）

从软件架构的角度看，jQuery 将 link 元素的事件发布给了任何想订阅此事件的人。这正是称其为 PubSub 模式的原因。

在老式 DOM 的事件 API 中，绑定至事件意味着要编写 object.onevent=...这样的代码，但现在它差不多被人忘光了，人们都转投至 PubSub 的怀抱了。Node 的 API 架构师因为太喜欢 PubSub，所以决定包含一个一般性的 PubSub 实体。这个实体叫做 EventEmitter（事件发生器），其他对象可以继承它。Node 中几乎所有的 I/O 源都是 EventEmitter 对象：文件流、HTTP 服务器，甚至是应用进程本身。以下例为证。

Distributed/processExit.js

```
['room', 'moon', 'cow jumping over the moon']
.forEach(function(name) {
  process.on('exit', function() {
    console.log('Goodnight, ' + name);
  });
});
```

浏览器端存在着无数的单机版 PubSub 库。此外，很多 MVC 框架，如 Backbone.js 和 Spine，都提供了自己的类 EventEmitter 模块。本章稍后再更详细地讨论 Backbone。

① 参见 http://api.jquery.com/on/。

2.1.1　EventEmitter对象

我们用 Node 的 EventEmitter 对象作为 PubSub 接口的例子。EventEmitter 有着简单而近乎最简化的设计。

要想给 EventEmitter 对象添加一个事件处理器，只要以事件类型和事件处理器为参数调用 on 方法即可。

```
emitter.on('evacuate', function(message) {
  console.log(message);
});
```

emit（意为“触发”）方法负责调用给定事件类型的所有处理器。举个例子，下面这行代码：

```
emitter.emit('evacuate');
```

将调用 evacuate 事件的所有处理器。

请注意，这里的术语事件跟事件队列没有任何关系。请参阅 2.1.3 节。

使用 emit 方法触发事件时，可以添加任意多的附加参数。所有参数均传递至所有处理器。

```
emitter.emit('evacuate', 'Woman and children first!');
```

事件名称不存在任何限制，然而 Node 相关文档还是规定了一条有用的约定。

> 通常，事件名称会表示为一个驼峰式大小写混合的字符串。[①]

① 参见 http://nodejs.org/docs/latest/api/events.html。驼峰式大小写是一种命名惯例，目的是增强标识符的可辨识度和可读性，因写得高低错落如驼峰而得名，可以分成 camelCased、CamelCased 等多种形式。——译者注

EventEmitter 对象的所有方法都是公有的，但一般约定只能从 EventEmitter 对象的“内部”触发事件。也就是说，如果有一个对象继承了 EventEmitter 原型并使用了 `this.emit` 方法来广播事件，则不应该从这个对象之外的其他地方再调用其 `emit` 方法。

2.1.2 玩转自己的PubSub

PubSub 模式的实现如此简单，以至于用十几行代码就能建立自己的 PubSub 实现。对于支持的每种事件类型，唯一需要存储的状态值就是一个事件处理器清单。

```
PubSub = {handlers: {}}
```

需要添加事件监听器时，只要将监听器推入数组末尾即可（这意味着总是会按照添加监听器的次序来调用监听器）。

```
PubSub.on = function(eventType, handler) {
  if (!(eventType in this.handlers)) {
    this.handlers[eventType] = [];
  }

  this.handlers[eventType].push(handler);
  return this;
}
```

接着，等到触发事件的时候，再循环遍历所有的事件处理器。

```
PubSub.emit = function(eventType) {
  var handlerArgs = Array.prototype.slice.call(arguments, 1);
  for (var i = 0; i < this.handlers[eventType].length; i++) {
    this.handlers[eventType][i].apply(this, handlerArgs);
  }
  return this;
}
```

就是这么简单！现在只实现了 Node 之 EventEmitter 对象的核心部分。（还没实现的重要部分只剩下移除事件处理器及附加一次性事件处理器等功能。）

当然，各种 PubSub 实现在特性方面会稍有不同。jQuery 团队注意到 jQuery 库里到处都在用几个不同的 PubSub 实现，于是决定在 jQuery 1.7 中将它们抽象为`$.Callbacks`[①]。这样就不再用数组来存储各种事件类型对应的事件处理器，而可以转用`$.Callbacks` 实例。

很多 PubSub 实现负责解析事件字符串以提供一些特殊功能。举个例子，你也许熟悉 jQuery 的名称空间化事件：如果绑定了名称为`"click.tbb"`和`"hover.tbb"`的两个事件，则简单地调用`unbind(".tbb")`就可以同时解绑定它们。Backbone.js 允许向`"all"`事件类型绑定事件处理器，这样不管发生什么事，都会导致这些事件处理器的触发。jQuery 和 Backbone.js 都支持用空格隔开多个事件来同时绑定或触发多种事件类型，譬如`"keypress mousemove"`。

2.1.3　同步性

尽管 PubSub 模式是一项处理异步事件的重要技术，但它内在跟异步没有任何关系。请考虑下面这段代码：

```
$('input[type=submit]')
.on('click', function() { console.log('foo'); })
.trigger('click');
console.log('bar');
```

这段代码的输出为：

① 参见 http://api.jquery.com/jQuery.Callbacks/。

```
❮foo
bar
```

这证明了 `click` 事件的处理器因 `trigger` 方法而立即被激活。事实上，只要触发了 jQuery 事件，就会不被中断地按顺序执行其所有事件处理器。

好吧，我们要明确一点：用户点击 Submit（提交）按钮时，这确实是一个异步事件。点击事件的第一个处理器会从事件队列中被触发。然而，事件处理器本身无法知道自己是从事件队列中还是从应用代码中运行的。

如果事件按顺序触发了过多的处理器，就会有阻塞线程且导致浏览器不响应的风险。更糟糕的是，如果事件处理器本身触发了事件，还很容易造成无限循环。

```
$('input[type=submit]')
.on('click', function() {
  $(this).trigger('click');  //堆栈上溢!
});
```

回想本章开头提到的文字处理程序的例子。用户按键时，需要发生很多事情，其中某些事还需要复杂的计算。全部做完这些事之后再返回事件队列，只会制造出响应迟钝的应用。

这个问题有一个很好的解决方案，就是对那些无需即刻发生的事情维持一个队列，并使用一个计时函数定时运行此队列中的下一项任务。首次尝试编码的结果可能像这样：

```
var tasks = [];
setInterval(function() {
  var nextTask;
```

```
  if (nextTask = tasks.shift()) {
    nextTask();
  };
}, 0);
```

（请参阅 4.4 节，了解一种更复杂精妙的作业排队技术。）

PubSub 模式简化了事件的命名、分发和堆积。任何时刻，只要直觉上认为对象会声明发生什么事情，就可以使用 PubSub 这种很棒的模式。

2.2 事件化模型

只要对象带有 PubSub 接口，就可以称之为事件化对象。特殊情况出现在用于存储数据的对象因内容变化而发布事件时，这里用于存储数据的对象又称作模型。模型就是 MVC（Model-View-Controller，模型–视图–控制器）中的那个 M。MVC 三层架构设计模式在最近几年里已经成为 JavaScript 编程中最热点的主题之一。MVC 的核心理念是应用程序应该以数据为中心，所以模型发生的事件会影响到 DOM（即 MVC 中的视图）和服务器（通过 MVC 中的控制器而产生影响）。

我们先来看看人气爆棚的 Backbone.js 框架[①]。可以像这样创建一个新的 Model（模型）对象：

```
style = new Backbone.Model(
  {font: 'Georgia'}
);
```

model 作为参数时只是代表了那个简单的可以传递的 JSON 对象。

---

① 参见 http://documentcloud.github.com/backbone/。

```
style.toJSON() // {"font": "Georgia"}
```

但不同于普通对象的是，这个 model 对象会在发生变化时发布通知。

```
style.on('change:font', function(model, font) {
  alert('Thank you for choosing ' + font + '!');
});
```

老式的 JavaScript 依靠输入事件的处理器直接改变 DOM。新式的 JavaScript 先改变模型，接着由模型触发事件而导致 DOM 的更新。在几乎所有的应用程序中，这种关注层面的分离都会带来更优雅、更直观的代码。

2.2.1 模型事件的传播

作为最简形式，MVC 三层架构只包括相互联系的模型和视图：“如果模型是这样变化的，那么 DOM 就要那样变化。”不过，MVC 三层架构最大的利好出现在 change（变化）事件冒泡上溯数据树的时候。不用再去订阅数据树每片叶子上发生的事件，而只需订阅数据树根和枝处发生的事件即可。

事件化模型的set/get方法

正如我们知道的，JavaScript确实没有一种每当对象变化时就触发事件的机制。因此请记住，事件化模型要想工作的话，必须要使用一些像Backbone.js之set/get这样的方法。

```
style.set({font: 'Palatino'});     // 触发器警报!
style.get('font');                 // 结果为"Palatino"
style.font = 'Comic Sans';         // 未触发任何事件
style.font;                        // 结果为"Comic Sans"
style.get('font');                 // 结果仍为"Palatino"
```

将来也许无需如此，前提是名为`Object.observe`的ECMAScript提案已经获得广泛接纳。[a]

a. 参见https://plus.google.com/111386188573471152118/posts/6peb6yffyWG。

为此，Backbone 的 Model 对象常常组织成 Backbone 集合的形式，其本质是事件化数组。我们可以监听什么时候对这些数组增减了 Model 对象。Backbone 集合可以自动传播其内蕴 Model 对象所发生的事件。

举个例子，假设有一个 `spriteCollection`（精灵集合）集合对象包含了上百个 Model 对象，这些 Model 对象代表了要画在 `canvas`（画布）元素上的一些东西。每当任意一个精灵发生变化，都需要重新绘制画布。我们不用逐个在那些精灵上附加 `redraw`（重绘）函数作为 `change` 事件的处理器，相反，只要写这样一行代码：

```
spriteCollection.on('change', redraw);
```

注意，集合事件的这种自动传播只能下传一层。Backbone 没有嵌套式集合这样的概念。不过，我们可以自行用 Backbone 的 `trigger` 方法来实现嵌套式集合的多层传播。有了多层传播机制之后，任意的 Backbone 对象都可以触发任意的事件。

2.2.2　事件循环与嵌套式变化

从一个对象向另一个对象传播事件的过程提出了一些需要关注的问题。如果每次有个对象上的事件引发了一系列事件并最终对这个对象本身触发了相同的事件，则结果就是*事件循环*。如果这种事件循环还是同步的，那就造成了堆栈上溢，就像我们在 2.1.3 节中看到的一样。

然而在很多时候，变化事件的循环恰恰是我们想要的。最常见的情况

就是双向绑定——两个模型的取值会彼此关联。假设我们想保证 x 始终等于 2 * y。

```
var x = new Backbone.Model({value: 0});
var y = new Backbone.Model({value: 0});
x.on('change:value', function(x, xVal) { y.set({value: xVal / 2}); });
y.on('change:value', function(y, yVal) { x.set({value: 2 * yVal}); });
```

你可能觉得当 x 或 y 的取值变化时，这段代码会导致无限循环。但实际上它相当安全，这要感谢 Backbone 中的两道保险。

- 当新值等于旧值时，set 方法不会导致触发 change 事件。
- 模型正处于自身的 change 事件期间时，不会再触发 change 事件。

第二道保险代表了一种自保哲学。假设模型的一个变化导致同一个模型又一次变化。由于第二次变化被“嵌套”在第一次变化内部，所以这次变化的发生悄无声息。外面的观察者没有机会回应这种静默的变化。

很明显，在 Backbone 中维持双向数据绑定是一个挑战。而另一个重要的 MVC 框架，即 Ember.js，采用了一种完全不同的方式：双向绑定必须作显式声明。一个值发生变化时，另一个值会通过延时事件作异步更新。于是，在触发这个异步更新事件之前，应用程序的数据将一直处于不一致的状态。

多个事件化模型之间的数据绑定问题不存在简单的解决方案。在 Backbone 中，有一种审慎绕过这个问题的途径就是 silent 标志。如果在 set 方法中添加了{silent:true}选项，则不会触发 change 事件。因此，如果多个彼此纠结的模型需要同时进行更新，一个很好的解决方法就是悄无声息地设置它们的值。然后，当这些模型的状态已

经一致时，才调用它们的 change 方法以触发对应的事件。

事件化模型为我们带来了一种将应用状态变化转换为事件的直观方式。Backbone 及其他 MVC 框架做的每件事都跟这些模型有关，这些模型的状态变化会触发 DOM 和服务器进行更新。要想掌控客户端 JavaScript 应用程序与日俱增的复杂度，运用事件化模型存储互斥数据是伟大长征的第一步。

2.3　jQuery 自定义事件

自定义事件是 jQuery 被低估的特性之一，它简化了强大分布式事件系统向任何 Web 应用程序的移植，而且无需额外的库。在 jQuery 中，可以使用 trigger 方法基于任意 DOM 元素触发任何想要的事件。

```
$('#tabby, #socks').on('meow', function() {
  console.log(this.id + ' meowed');
});
$('#tabby').trigger('meow'); // "tabby meowed"
$('#socks').trigger('meow'); // "socks meowed"
```

如果以前用过 DOM 事件，则肯定熟悉冒泡技术。只要某个 DOM 元素触发了某个事件（譬如'click'事件），其父元素就会接着触发这个事件，接着是父元素的父元素，以此类推，一直上溯到根元素（即 document），除非在这条冒泡之路的某个地方调用了事件的 stopPropagation 方法。（如果事件处理器返回 false，则 jQuery 会替我们自动调用 stopPropagation 方法。）但你是否也知道 jQuery 自定义事件的冒泡技术呢？举个例子，假设有个名称为“soda”的 span 元素嵌套在名称为“bottle”的 div 元素中，代码如下。

```
$('#soda, #bottle').on('fizz', function() {
  console.log(this.id + ' emitted fizz');
});
$('#soda').trigger('fizz');
```

得到的输出如下：

```
soda emitted fizz
bottle emitted fizz
```

这种冒泡方式并非始终受人欢迎，从下面的 tooltip（工具提示条）示例就能看到这一点。幸运的是，jQuery 同样提供了非冒泡式的 `triggerHandler` 方法。

示例：工具提示条

事件直观映射至页面元素之后，jQuery 就成为分发这些事件的一种理想方式。举个例子，假设正在编写一个关于工具提示条的库，并希望任一时刻只能看到一个工具提示条。我们可能会简单地写下这行代码：

```
$('.tooltip').remove();
```

并将它添加到那个负责新添工具提示条的函数的开头。但后来我们又想独立出某些容器（譬如，当侧边栏显示新的工具提示条时，其他地方的工具提示条不受影响），该怎么办？如果反过来，又该怎么办？编写一个选择器来选中“那些类别为 `tooltip` 且不是 `sidebar` 后代的 DOM 元素”，这显然很复杂，而且会非常没有效率。如果允许独立容器作任意深度的嵌套，这个问题的难度还会呈指数级增长。

不过，利用事件逻辑而不是选择器逻辑来实现这一行为则会很容易。

```
// $container could be $('#sidebar') or $(document)
$container.triggerHandler('newTooltip');
$container.one('newTooltip', function() {
  $tooltip.remove();
});
```

（请注意这里使用了 jQuery 的 one 来代替 on。这两者的区别在于，one 在触发处理器之后会自动将其删除。）

有了这两行代码之后，所有的工具提示条都会监听自己的容器，并且每当容器获得新工具提示条时就移除自己。这种优美、直接、有效的方式拯救了我们，让我们无需存储任何状态，也不用鼓弄那些复杂的选择器。（复杂选择器会让那些旧式浏览器慢得像乌龟爬——在不遍历整个文档的情况下，IE7 及更早版本甚至都无法选中类别为 tooltip 的所有 DOM 元素！）

请注意，实际上在这个案例中事件冒泡技术会破坏我们的意图：希望在侧边栏新建一个工具提示条时，只有那些监听侧边栏之 'newTooltip' 事件的工具提示条消失不见，而那些监听外围 document 元素的工具提示条不受影响。请始终认真思考这个问题：trigger 或 triggerHandler，到底哪一个才是完成该任务的合适工具？

jQuery 自定义事件是 PubSub 模式的忤逆产物，因为这里由可选择的 DOM 元素而不是脚本中的对象来触发事件。事件化模型更像是一种直观表达状态相关事件的方式，而 jQuery 的自定义事件允许直接通过 DOM 来表达 DOM 相关的事件，不必再把 DOM 变化的状态复制到应用程序的其他地方。请大方地取用这些特性，但要尽量避免依赖应用程序的标记语言结构——你肯定不希望下次修改设计的时候破坏自己的脚本。

2.4　小结

在本章中，我们学到了 PubSub 如何作为最基本的 JavaScript 设计模式之一实现了分布式事件。如果没有订阅那些行将发布的事件，PubSub 将是完全隐形的。正确运用 PubSub 模式的关键是判定由哪些实体分发事件。

我们已经看到，任何对象只要简单地继承诸如 Node 之 `EventEmitter` 这样的原型，就可以用作 PubSub 实体。当对象关联着一组异步任务或一系列 I/O 事件时，把它变成事件化对象会是个不错的想法。

有一类特殊的事件化对象是 MVC 库（如 Backbone.js）中的模型。这些模型既包含着应用程序的状态数据，又能声明自己发生的变化。`change` 事件可以触发应用程序的逻辑，引起 DOM 的更新，并导致服务器的同步。所有这些都是自然而然发生的，这也解释了 Backbone 为什么会成为火爆异常的 JavaScript 库之一。

我们还看到，jQuery 不仅能很好地响应浏览器提供的 DOM 事件，而且还非常适用于那些与 DOM 元素变化有关的分布式事件。事件化的对象与 DOM 元素的事件可以彼此完美互补，这有助于保持应用状态数据与应用视图相对于彼此的封装状态。

所有这些都是活生生的 PubSub 实例。不过，尽管 PubSub 模式如此多才多艺，但它不是适用于所有任务的万能工具。PubSub 模式尤其不适用于一次性事件，一次性事件要求对异步函数执行的一次性任务的两种结果（完成任务或任务失败）做不同的处理。（Ajax 请求就是常见的一次性事件实例。）用于解决一次性事件问题的工具叫做 Promise，这正是下一章的主题。

第 3 章

Promise 对象和 Deferred 对象

2010 年，我和我那能者多劳的 Ajax 同事有过这样一次对话。

我：嗨，您能帮我从这个 URL 抓取一些数据回来吗?

Ajax 大拿：我这就去搞定它！不过你要给我一个 `success` 回调，好让我搞定后就通知你。

我：好，给您。十分感谢。

Ajax 大拿：哦，对了，你还要给我一个 `error` 回调。你知道啦，以防万一嘛！

我：说得不错。还要其他东西吗?

Ajax 大拿：嘿，我发现你给的那两个回调之间有一些代码重复呀！你可以把那些重复的代码移到第三个回调里面，就叫做 `always` 回调好了。

我：（不耐烦）好吧好吧，我去把它们重构一下。另外，给您提点意见呀，您干嘛不先去干活儿呢？我稍后再给您那些回调不行吗?

Ajax 大拿：（大怒）那我变成什么了？`EventEmitter` 对象吗?

幸好，jQuery 1.5 改变了 Ajax 大拿那种“马上就要”的态度。我们知道和喜欢的所有 Ajax 函数（`$.ajax`、`$.get` 及`$.post`）现在都会返回 Promise（承诺）对象。Promise 对象代表一项有两种可能结果（成功或失败）的任务，它还持有多个回调，出现不同结果时会分别触发相应的回调。举个例子，jQuery 1.4 中的代码必须写成这样：

Promises/get-1.4.js

```
$.get('/mydata', {
  success: onSuccess,
  failure: onFailure,
  always: onAlways
});
```

而到了 jQuery 1.5+，可以写成这样：

Promises/get-1.5.js

```
var promise = $.get('/mydata');
promise.done(onSuccess);
promise.fail(onFailure);
promise.always(onAlways);
```

大家可能会奇怪:这种变化能有什么好处呢？为什么非得在触发 Ajax 调用之后再附加回调呢？一言以蔽之：封装。如果 Ajax 调用要实现很多效果（既要触发动画，又要插入 HTML，还要锁定/解锁用户输入，等等），那么仅由负责发出请求的那部分应用代码来处理所有这些效果，显然很蠢很拙劣。

只传递 Promise 对象就会优雅得多。传递 Promise 对象就相当于声明：“你感兴趣的某某事就要发生了。想知道什么时候完事吗？给这个 Promise 对象一个回调就行啦！”Promise 对象也和 `EventEmitter` 对象一样，允许向同一个事件绑定任意多的处理器（*堆积技术*）。对于多个 Ajax 调用分享某个功能小片段（譬如“正加载”动画）的情况，

堆积技术也会使降低代码重复度容易很多。

不过使用 Promise 对象的最大优势仍然在于，它可以轻松从现有 Promise 对象派生出新的 Promise 对象。我们可以要求代表着并行任务的两个 Promise 对象合并成一个 Promise 对象，由后者负责通知前面那些任务都已完成。也可以要求代表着任务系列中首任务的 Promise 对象派生出一个能代表任务系列中末任务的 Promise 对象，这样后者就能知道这一系列任务是否均已完成。待会儿我们就会看到，Promise 对象天生就适合用来进行这些操作。

3.1 Promise 极简史

Promise 对象曾经以多种形式存在于很多语言中。这个词最先由 C++ 工程师用在 Xanadu 项目中，Xanadu 项目是 Web 应用项目的先驱。随后 Promise 被用在 E 编程语言中，这又激发了 Python 开发人员的灵感，将它实现成了 Twisted 框架的 Deferred 对象。

2007 年，Promise 赶上了 JavaScript 大潮，那时 Dojo 框架刚从 Twisted 框架汲取灵感，新增了一个叫做 `dojo.Deferred` 的对象。也就在那个时候，相对成熟的 Dojo 框架与初出茅庐的 jQuery 框架激烈地争夺着人气和名望。2009 年，Kris Zyp 有感于 `dojo.Deferred` 的影响力提出了 CommonJS 之 Promises/A 规范[①]。同年，Node.js 首次亮相。Node 早期的几个版本在其非阻塞式 API 中用到了 Promise。但到了 2010 年 2 月，Ryan Dahl 决定切换至当时为人所熟知的 `callback(err, result...)`格式，因为 Promise 是一种属于“用户之境”的

① 参见 http://wiki.commonjs.org/wiki/Promises/A。

甚高层构造。

Ryan Dahl 的决定为那些以 Node 为竞争目标的 Promise 实现腾出了舞台，其中就有著名的 Kris Kowal 的 Q.js[①]和 AJ ONeal 的 Futures[②]。（在一般性用法中，Promise、Deferred 和 Future 这三个词大体可算作同义词。）Q.js 是 Promises/A 规范的一种相当直观的实现。Futures 是一种更广泛的工具集，结合了很多在其他库（如 Async.js）中才能找到的工作流控制特性。

不过，Promise 今天受到如此多关注的原因当然是 jQuery。jQuery 1.5 在 2011 年 1 月携`$.ajax` 重量级重写之势，用其 Promise 实现震惊了无数初次接触 Promise 对象的开发者。不过，其他的开发者则忧心忡忡，因为 jQuery 1.5 对 Promises/A 规范的无视导致了微妙的 API 差异。

除了 3.7 节之外，本章其余部分的关注点都是 jQuery 的 Promise 实现。此外还会讲述 jQuery 在用语上的不同之处，特别是两点，一是 Deferred 与 Promise 之间的区别（下一节就会看到），二是 resolve 用作 reject 的反义词。

3.2　生成 Promise 对象

本章一开始就展示了如何用 jQuery 1.5+中的 Ajax 方法（`$.ajax`、`$.get` 及`$.post`）返回 Promise 对象。但要想真正理解 Promise，我们需要自己动手生成它。

① 参见 https://github.com/kriskowal/q。
② 参见 https://github.com/coolaj86/futures。

假设我们提示用户应敲击 Y 键或 N 键。为此要做的第一件事就是生成一个`$.Deferred`实例以代表用户做出的决定。

```
var promptDeferred = new $.Deferred();
promptDeferred.always(function(){ console.log('A choice was made:'); });
promptDeferred.done(function(){ console.log('Starting game...'); });
promptDeferred.fail(function(){ console.log('No game today.'); });
```

（注：`always`关键字仅适用于 jQuery 1.6+。）

大家可能会奇怪：为什么本节叫做“生成 Promise 对象”，却要生成一个 Deferred（延迟）实例？别担心，Deferred 就是 Promise！更准确地说，Deferred 是 Promise 的超集，它比 Promise 多了一项关键特性：可以直接触发。纯 Promise 实例只允许添加多个调用，而且必须由其他什么东西来触发这些调用。

使用`resolve`（执行）方法和`reject`（拒绝）方法均可触发 Deferred 对象。

```
$('#playGame').focus().on('keypress', function(e) {
  var Y = 121, N = 110;
  if (e.keyCode === Y) {
    promptDeferred.resolve();
  } else if (e.keyCode === N) {
    promptDeferred.reject();
  } else {
    return false; // 这里的 Deferred 对象保持着挂起状态
  };
});
```

请访问 http://jsfiddle.net/TrevorBurnham/PJ6Bf/查看这个例子的运行情况。加载页面，敲击 Y 键。控制台会这样说：

```
《A choice was made:
 Starting game...
```

大家看懂是怎么回事了吗？执行了 Deferred（即对 Deferred 对象调用了 `resolve` 方法）之后，即运行该对象的 `always`（恒常）回调和 `done`（已完成）回调。（会按照绑定回调的次序来运行回调，这可不是巧合哦！）

刷新页面，敲击 N 键。

```
《A choice was made:
 No game today.
```

这样，拒绝了 Deferred（即对 Deferred 对象调用了 `reject` 方法）之后，即运行该对象的 `always` 回调和 `fail`（失败）回调。注意，始终会按照绑定回调的次序来运行回调。如果最后绑定的是 `always` 回调，则控制台的输出行顺序会反过来。

再试着反复敲击 Y 键和 N 键。第一次做出选择之后，就再也没有反应了！这是因为 Promise 只能执行或拒绝一次，之后就失效了。我们断言，Promise 对象会一直保持挂起状态，直到被执行或拒绝。对 Promise 对象调用 `state`（状态）方法，可以查看其状态是`"pending"`、`"resolved"`，还是`"rejected"`。（到 jQuery 1.7 才添加了 `state` 方法，此前的版本使用的是 `isResolved` 和 `isRejected`。）

如果正在进行的一次性异步操作的结果可以笼统地分成两种（如成功/失败，或接受/拒绝），则生成 Deferred 对象就能直观地表达这次任务。

3.2.1　生成纯Promise对象

我们刚刚了解到 Deferred 对象也是 Promise 对象，那么，如何得到一

个不是 Deferred 对象的 Promise 对象呢？很简单，对 Deferred 对象调用 `promise` 方法即可。

两种说法，一个意思

大家可能已经注意到了，在执行或拒绝Promise的时候，我一直说的是“触发”Promise对象；已执行或已拒绝Promise则称Promise对象“已触发”。这种说法并不标准，不过本章仍会沿用。遗憾的是，jQuery除了拙劣地念叨“非挂起”之外，并没有一种简洁明了的说法来指代那些已执行或已拒绝的Promise对象。

CommonJS的Promises/A规范及其实现中使用了一种更合理的说法：Promise对象已履行（关键字为`fulfill`）或已拒绝，这两种情况都称Promise已执行。3.7节会对此作进一步阐述。

```
var promptPromise = promptDeferred.promise();
```

`promptPromise` 只是 `promptDeferred` 对象的一个没有 `resolve`/`reject` 方法的副本。我们把回调绑定至 Deferred 或其下辖的 Promise 并无不同，因为这两个对象本质上分享着同样的回调。它们也分享着同样的 `state`（返回的状态值为`"pending"`、`"resolved"`或`"rejected"`）。这意味着，对同一个 Deferred 对象生成多个 Promise 对象是毫无意义的。事实上，jQuery 给出的只不过是同一个对象。

```
var promise1 = promptDeferred.promise();
var promise2 = promptDeferred.promise();
console.log(promise1 === promise2); // true
```

而且，对一个纯 Promise 对象再调用 `promise` 方法，产生的只不过是一个指向相同对象的引用。

```
console.log(promise1 === promise1.promise()); // true
```

使用 promise 方法的唯一理由就是“封装”。如果传递 promptPromise 对象，但保留 promptDeferred 对象为己所用，则可以肯定的是，除非是你自己想触发那些回调，否则任何回调都不会被触发。

在此重申一点：每个 Deferred 对象都含有一个 Promise 对象，而每个 Promise 对象都代表着一个 Deferred 对象。有了 Deferred 对象，就可以控制其状态，而有了纯 Promise 对象，只能读取其状态及附加回调。

3.2.2　jQuery API中的Promise对象

本章开头列举了 jQuery 的 Ajax 函数（$.ajax、$.get 及$.post）可返回的几个 Promise 对象。Ajax 是演示 Promise 的绝佳用例：每次对远程服务器的调用都或成功或失败，而我们希望以不同的方式来处理这两种情况。不过，Promise 也同样适用于本地的一些异步操作，譬如动画。

在 jQuery 中，任何动画方法都可以接受传入的回调，以便在完成动画时发出通知。

```
$('.error').fadeIn(afterErrorShown);
```

在 jQuery 1.6+中，可以转而要求 jQuery 对象生成 Promise，后者代表了这个对象已附加动画的完成情况，即是否完成了目前正处于挂起状态的动画。

```
var errorPromise = $('.error').fadeIn().promise();
errorPromise.done(afterErrorShown);
```

对同一个 jQuery 对象附加的多个动画会排入队列按顺序运行。仅当

调用 `promise` 方法之时已入列的全部动画均已执行之后，相应的 Promise 对象才会执行。因此，这会产生两个不同的、按顺序执行的 Promise 对象（或者根本就不执行，若先调用 `stop` 方法的话）。

```
var $flash = $('.flash');
var showPromise = $flash.show().promise();
var hidePromise = $flash.hide().promise();
```

相当简单，对不对？在 jQuery 1.6 及 jQuery 1.7 中，jQuery 对象的 `promise` 方法只是一种权宜之计。如果使用 Deferred 对象的 `resolve` 方法作为动画的回调，即可自行轻松生成一个行为完全相同的动画版 Promise 对象。

```
var slideUpDeferred = new $.Deferred();
$('.menu').slideUp(slideUpDeferred.resolve);
var slideUpPromise = slideUpDeferred.promise();
```

在本书付梓之前刚刚发布的 jQuery 1.8 中，动画版 Promise 已变成更加强大的对象。动画版 Promise 附加了额外的信息，其中包括动画运行过程中的计算值 `props`（这对调试非常有价值）。此外，对动画版 Promise 对象，还可以获得进度通知（请参阅 3.4 节），以及即时调整动画。有关这些新特性的文档草稿可见于 https://gist.github.com/54829d408993526fe475。

jQuery 1.8 又向 jQuery 大家庭中新添了一种 Promise 资源：`$.ready.promise()`也能生成一个 Promise 对象，并且当文档就绪时即执行该对象。这意味着以下 3 行代码现在是等效的。

```
$(onReady);
 $(document).ready(onReady);
 $.ready.promise().done(onReady);
```

本节介绍了如何获得 jQuery 中的 Promise 对象：或者生成一个 `$.Deferred` 实例（这会带来一个可自行控制的 Promise），或者进行一次可返回 Promise 对象的 API 调用。以下几节将介绍我们能对这些 Promise 对象做些什么。

3.3　向回调传递数据

Promise 对象可以向其回调提供额外的信息。举个例子，下面这两个 Ajax 代码片段是等效的。

```
// 直接使用回调
$.get(url, successCallback);

// 将回调绑定至 Promise 对象
var fetchingData = $.get(url);
fetchingData.done(successCallback);
```

执行或拒绝 Deferred 对象时，提供的任何参数都会转发至相应的回调。

```
var aDreamDeferred = new $.Deferred();
aDreamDeferred.done(function(subject) {
  console.log('I had the most wonderful dream about', subject);
});
aDreamDeferred.resolve('the JS event model');
```

```
❮I had the most wonderful dream about the JS event model
```

还有一些特殊的方法能实现在特定上下文中运行回调（即将 `this` 设置为特定的值）：`resolveWith` 和 `rejectWith`。此时只需传递上下文环境作为第一个参数，同时以数组的形式传递所有其他参数。

```
var slashdotter = {
  comment: function(editor){
    console.log('Obviously', editor, 'is the best text editor.');
  }
};
var grammarDeferred = new $.Deferred();
grammarDeferred.done(function(verb, object) {
  this[verb](object);
});
grammarDeferred.resolveWith(slashdotter, ['comment', 'Emacs']);
```

```
❮Obviously Emacs is the best text editor.
```

然而，将参数打包成数组是很痛苦的。所以还有一个小窍门：不再使用 `resolveWith/rejectWith`，而是直接在目标上下文中调用 `resolve/reject` 方法，这是因为 `resolve/reject` 可以直接将其上下文环境传递至自己所触发的回调。因此，对于前面那个例子，使用以下代码亦可得到同样的结果。

```
grammarDeferred.resolve.call(slashdotter, 'comment', 'Emacs');
```

3.4　进度通知

Promise 对象是你希望任务结束时发生的一些事。但是，你没听说过过程和结果同样重要吗？难道你不以为然吗？

幸好，jQuery 团队意识到了这一点并遵守 Promises/A 规范，于是在 jQuery 1.7 中为 Promise 对象新添了一种可以调用无数次的回调。这个回调叫做 `progress`（进度）。举个例子，假设有人正在奋力达成美国全国小说写作月（National Novel Writing Month，简写为

NaNoWriMo）[1]项目设定的日均码字目标，而我们希望更新一个指示器以反映他距离实现这个目标还有多远。

```
var nanowrimoing = $.Deferred();
var wordGoal = 5000;
nanowrimoing.progress(function(wordCount) {
  var percentComplete = Math.floor(wordCount / wordGoal * 100);
  $('#indicator').text(percentComplete + '% complete');
});
nanowrimoing.done(function(){
  $('#indicator').text('Good job!');
});
```

Deferred 对象的 nanowrimoing 准备就绪之后，可以像下面这样对字数的变化作出响应。

```
$('#document').on('keypress', function(){
  var wordCount = $(this).val().split(/\s+/).length;
  if (wordCount >= wordGoal) {
    nanowrimoing.resolve();
  };
  nanowrimoing.notify(wordCount);
});
```

Deferred 对象的 notify（通知）调用会调用我们设定的 progress 回调。就像 resolve 和 reject 一样，notify 也能接受任意参数。请注意，一旦执行了 nanowrimoing 对象，则再作 nanowrimoing.notify 调用将不会有任何反应，这就像任何额外的 resolve 调用及 reject 调用也会被直接无视一样。

简单总结一下，Promise 对象接受 3 种回调形式：done、fail 和 progress。执行 Promise 对象时，运行的是 done 回调；拒绝 Promise

① 参见 http://www.nanowrimo.org/。

对象时，运行的是 `fail` 回调；对处于挂起状态的 Deferred 对象调用 `notify` 时，运行的是 `progress` 回调。

3.5 Promise 对象的合并

进度通知的存在并没有改变每个 Promise 对象的最终状态为已执行或已拒绝这一事实。（否则，Promise 对象将永远保持挂起状态。）但为什么要这样呢？为什么不让 Promise 对象随时变化成任意的状态，而偏偏只有这两种状态呢？

这样设计 Promise，其主要原因是程序员一直都在跟二进制打交道。我们这些码农非常清楚如何把 1 和 0 揉捏起来建成令人瞠目的逻辑之塔。Promise 如此强大的一个主要原因是，它允许我们把任务当成布尔量来处理。

Promise 对象的逻辑合并技术有一个最常见的用例：判定一组异步任务何时完成。假设我们正在播放一段演示视频，同时又在加载服务器上的一个游戏。我们希望这两件事一旦结束（对次序没有要求），就马上启动游戏。

- 演示视频已经播放完毕。
- 游戏已经加载完毕。

这两个进程各用一个 Promise 对象来表示，我们的任务就是在这两个 Promise 均已执行时启动游戏。我们如何做到这一点呢？

下面隆重介绍 jQuery 的 `when` 方法！

```
var gameReadying = $.when(tutorialPromise, gameLoadedPromise);
gameReadying.done(startGame);
```

when 相当于 Promise 执行情况的逻辑与运算符（AND）。一旦给定的所有 Promise 均已执行，就立即执行 when 方法产生的 Promise 对象；或者，一旦给定的任意一个 Promise 被拒绝，就立即拒绝 when 产生的 Promise。

when 方法的绝佳用例是合并多重 Ajax 调用。假设需要马上进行两次 post 调用，而且要在这两次调用都成功时收到通知，这时就无需再为每次调用请求分别定义一个回调。

```
$.when($.post('/1', data1), $.post('/2', data2))
.then(onPosted, onFailure);
```

调用成功时，when 可以访问下辖的各个成员 Promise 对象的回调参数，不过这么做很复杂。这些回调参数会当作参数列表进行传递，传递的次序和成员 Promise 对象传递给 when 方法时一样。如果某个成员 Promise 对象提供多个回调参数，则这些参数会先转换成数组。

因此，要想根据赋予$.when 方法的所有成员 Promise 对象获得全部回调参数，可能会写出像下面这样的代码（但笔者并不推荐这么做）。

```
$.when(promise1, promise2)
.done(function(promise1Args, promise2Args) {
  // ...
});
```

在这个例子中，如果执行 promise1 时用到了一个参数'complete'，执行 promise2 时用到了 3 个参数（1、2、3），则 promise1Args 就是字符串'complete'，promise2Args 就是数组[1,2,3]。

虽然有可能，但如果不是绝对必要，我们不应该自行解析 `when` 回调的参数，相反应该直接向那些传递至 `when` 方法的成员 Promise 对象附加回调来收集相应的结果。

```
var serverData = {};
var getting1 = $.get('/1')
.done(function(result) {serverData['1'] = result;});
var getting2 = $.get('/2')
.done(function(result) {serverData['2'] = result;});
$.when(getting1, getting2)
.done(function() {
  // 获得的信息现在都已位于serverData……
});
```

函数的Promise用法

`$.when` 及其他能取用 Promise 对象的 jQuery 方法均支持传入非 Promise 对象作为参数。这些非 Promise 参数会被当成因相应参数位置已赋值而执行的 Promise 对象来处理。例如

```
$.when('foo')
```

会生成一个因赋值`'foo'`而立即执行的 Promise 对象。再譬如

```
var promise = $.Deferred().resolve('manchu');
$.when('foo', promise)
```

会生成一个因赋值`'foo'`和`'manchu'`而立即执行的 Promise 对象。代码

```
var promise = $.Deferred().resolve(1, 2, 3);
$.when('test', promise)
```

会生成一个因赋值`'test'`和数组`[1,2,3]`而立即执行的 Promise 对

象。（请记住，Deferred 对象传递多个参数给 `resolve` 方法时，`$.when` 会把这些参数转换成一个数组。）

这带来了一个问题：`$.when` 如何知道参数是不是 Promise 对象呢？答案是：jQuery 负责检查 `$.when` 的各个参数是否带有 `promise` 方法，如果有就使用该方法返回的值。Promise 对象的 `promise` 方法会直接返回自身。

正如 3.2.2 节所述，jQuery 对象也可以有 `promise` 方法，这意味着 `$.when` 方法强行将那些带 `promise` 方法的 jQuery 对象转换成了 jQuery 动画版 Promise 对象。因此，如果想生成一个在抓取某些数据且已完成`#loading` 动画之后执行的 Promise 对象，只需写下下面这样的代码：

```
var fetching = $.get('/myData');
$.when(fetching, $('#loading'));
```

只是请记住，必须要在动画开始之后再执行`$.when` 生成的那个 Promise 对象。如果`#loading` 的动画队列为空，则立即执行相应的 Promise 对象。

3.6 管道连接未来

在 JavaScript 中常常无法便捷地执行一系列异步任务，一个主要原因是无法在第一个任务结束之前就向第二个任务附加处理器。举个例子，假设我们要从一个 URL 抓取数据（`GET`），接着又将这些数据发送给另一个 URL（`POST`）。

```
var getPromise = $.get('/query');
getPromise.done(function(data) {
  var postPromise = $.post('/search', data);
});
// 现在我们想给 postPromise 附加处理器……
```

看到这里的问题了吗？在 GET 操作成功之前我们无法对 postPromise 对象绑定回调，因为这时 postPromise 对象还不存在！除非我们已经得到因$.get 调用而异步抓取的数据，否则甚至无法进行那个负责生成 postPromise 对象的$.post 调用！

这正是 jQuery 1.6 为 Promise 对象新增 pipe(管道)方法的原因。pipe 好像在说：“请针对这个 Promise 对象给我一个回调，我会归还一个 Promise 对象以表示回调运行的结果。”

```
var getPromise = $.get('/query');
var postPromise = getPromise.pipe(function(data) {
  return $.post('/search', data);
});
```

看起来就像黑魔法，对吧？下面是详情大揭秘：pipe 最多能接受 3 个参数，它们对应着 Promise 对象的 3 种回调类型：done、fail 和 progress。也就是说，我们在上述例子中只提供了执行 getPromise 时应运行的那个回调。当这个回调返回的 Promise 对象已经执行/拒绝时，pipe 方法返回的那个新 Promise 对象也就可以执行/拒绝。

从效果上看，pipe 就是通向未来的一扇窗户！

我们也可以通过修改 pipe 回调参数来“滤清”Promise 对象。如果 pipe 方法的回调返回值不是 Promise/Deferred 对象，它就会变成回调参数。举例来说，假设有个 Promise 对象发出的进度通知表示成 0 与 1 之间的某个数，则可以使用 pipe 方法生成一个完全相同的 Promise

对象，但它发出的进度通知却转变成可读性更高的字符串。

```
var promise2 = promise1.pipe(null, null, function(progress) {
  return Math.floor(progress * 100) + '% complete';
});
```

总的说来，pipe 的回调可以做以下两件事情。

- 如果 pipe 回调返回的是 Promise 对象，则 pipe 生成的那个 Promise 对象会模仿这个 Promise 对象。
- 如果 pipe 回调返回的是非 Promise 对象（值或空白），则 pipe 生成的那个 Promise 对象会立即因该赋值而执行、拒绝或得到通知，具体取决于调用 pipe 的那个初始 Promise 对象刚刚发生了什么。

pipe 判定参数是否为 Promise 对象的方法和$.when 完全一样：如果 pipe 的参数带有 promise 方法，则该方法的返回值会被当作 Promise 对象以代表调用 pipe 的那个初始 Promise 对象。再重申一次，promise. promise() === promise。

管道级联技术

pipe 方法并不要求提供所有的可能回调。事实上，我们通常只想写成这样：

```
var pipedPromise = originalPromise.pipe(successCallback);
```

或是这样：

```
var pipedPromise = originalPromise.pipe(null, failCallback);
```

我们能看出初始 Promise 对象（即 originalPromise）在成功/失败之后应触发的回调（第一种情况下因任务成功而触发 successCallback，

第二种情况下因任务失败而触发 failCallback），所以管道末尾的 Promise（即 pipedPromise）行为取决于 successCallback/failCallback 的返回值。但是，如果并没有在 pipe 方法中为初始 Promise 的任务结果指定回调，又该怎么办呢？

很简单，管道末尾的 Promise 在这些情况下直接模仿那个初始 Promise 对象。我们可以这样说：初始 Promise 对象的行为一直级联到管道末尾的 Promise 对象。这种级联技术非常有用，因为它让我们不费吹灰之力就能定义异步任务的分化逻辑。假设有这样一个分成 3 步走的进程。

```
var step1 = $.post('/step1', data1);
var step2 = step1.pipe(function() {
  return $.post('/step2', data2);
});
var lastStep = step2.pipe(function() {
  return $.post('/step3', data3);
});
```

这里的 lastStep 对象当且仅当所有这 3 个 Ajax 调用都成功完成时才执行，其中任意一个 Ajax 调用未能成功完成，lastStep 均被拒绝。如果只在乎整体进程，则可以省略掉前面的变量声明。

```
var posting = $.post('/step1', data1)
  .pipe(function() {
    return $.post('/step2', data2);
  })
    .pipe(function() {
      return $.post('/step3', data3);
    });
```

也可以让后面的 pipe 嵌套在前面那个 pipe 的里面。

```
var posting = $.post('/step1', data1)
  .pipe(function() {
    return $.post('/step2', data2)
    .pipe(function() {
      return $.post('/step3', data3);
    });
  });
```

当然，这会重现金字塔厄运。大家应该了解这种书写风格，不过请尽量逐一声明 `pipe` 生成的那些 Promise 对象。也许并不需要这些变量名称，但它们能让代码更加自文档化。

我们的 jQuery Promise 之旅到此就结束了。接下来简单介绍一下其主要替代方案：CommonJS 的 Promises/A 规范及其旗舰版实现 Q.js。

3.7 jQuery 与 Promises/A 的对比

从功能上看，jQuery 的 Promise 与 CommonJS 的 Promises/A 几乎完全一样。Q.js 库是最流行的 Promises/A 实现，其提供的方法甚至能与 jQuery 的 Promise 和谐共存。这两者的区别只是形式上的，即用相同的词语表示不同的含义。

3.2 节中提到过，jQuery 使用 resolve 作为 fail 的反义词，而 Promises/A 使用的是 fulfill。在 Promises/A 规范中，Promise 对象不管是已履行还是已失败，都称“已执行”。

在 jQuery 1.8 问世之前，jQuery 的 `then` 方法只是一种可以同时调用 `done`、`fail` 和 `progress` 这 3 种回调的速写法，而 Promises/A 的 `then` 在行为上更像是 jQuery 的 `pipe`。jQuery 1.8 订正了这个问题，使得 `then` 成为 `pipe` 的同义词。不过，由于向后兼容的问题，jQuery 的

Promise 再如何对 Promises/A 示好也不太会招人待见。

当然还有其他一些细微的差别。例如，在 Promises/A 规范中，由 `then` 方法生成的 Promise 对象是已执行还是已拒绝，取决于由 `then` 方法调用的那个回调是返回值还是抛出错误。（在 jQuery 的 Promise 对象的回调中抛出错误是个糟糕的主意，因为错误不会被捕获。）

基于上述原因，应该尽量避免在同一个项目中与多个 Promise 实现“打情骂俏”。如果因 jQuery 方法而得到 Promise 对象，请使用 jQuery 的 Promise。如果因使用其他库而得到 CommonJS Promise 对象，则请遵守 Promises/A 规范。而 Q.js 可以轻松 “消化” jQuery 的 Promise 对象。

```
var qPromise = Q.when(jqPromise);
```

只要这两套标准仍然存在差异，这就是让它们融洽相处的最好方法。更多信息请参阅 Q.js 文档。[①]

3.8　用 Promise 对象代替回调函数

理想情况下，开始执行异步任务的任何函数都应该返回 Promise 对象。遗憾的是，大多数 JavaScript API（包括所有浏览器及 Node.js 均使用的那些原生函数）都基于回调函数，而不是基于 Promise 对象。在本节中，我们将看到如何在基于回调函数的 API 中使用 Promise 对象。

在基于回调函数的 API 中使用 Promise 对象的最直接的方法是，生成一个 Deferred 对象并传递其触发器函数作为 API 的回调参数。例如，我们可以把 Deferred 对象的 `resolve` 方法传递给一个像 `setTimeout`

① 参见 https://github.com/kriskowal/q。

这样的简单异步函数。

```
var timing = new $.Deferred();
setTimeout(timing.resolve, 500);
```

考虑到 API 可能会出错，我们还要写一个根据情况指向 `resolve` 或 `reject` 的回调函数。例如，我们会这样写 Node 风格的回调：

```
var fileReading = new $.Deferred();
fs.readFile(filename, 'utf8', function(err) {
  if (err) {
    fileReading.reject(err);
  } else {
    fileReading.resolve(Array.prototype.slice.call(arguments, 1));
  };
});
```

（没错，确实能在 Node 中使用 jQuery。只要运行命令行 `npm install jquery` 即可，其用法和其他所有模块一样。还有一个库叫 Standalone Deferred，它独立实现了 jQuery 风格的 Promise。[①]）

总这么写很麻烦，所以何不写一个工具函数以根据任何给定 Deferred 对象来生成 Node 风格的回调呢？

```
deferredCallback = function(deferred) {
  return function(err) {
    if (err) {
      deferred.reject(err);
    } else {
      deferred.resolve(Array.prototype.slice.call(arguments, 1));
    };
  };
}
```

① 参见 https://github.com/Mumakil/Standalone-Deferred。

有了这个工具函数，前面那个例子就可以写成这样：

```
var fileReading = new $.Deferred();
fs.readFile(filename, 'utf8', deferredCallback(fileReading));
```

Q.js 的 Deferred 对象为此提供了一个现成的 `node` 方法：

```
var fileReading = Q.defer();
fs.readFile(filename, 'utf8', fileReading.node());
```

随着 Promise 越来越流行，会有越来越多的 JavaScript 库循着 jQuery 的脚步，要求其异步函数必须返回 Promise 对象。到那时，只需要几行代码就能将想用的任何异步函数转变成 Promise 对象的生成函数。

3.9 小结

在笔者看来,Promise 是过去几年中 jQuery 最激动人心的新特性之一。这不仅是因为 Promise 大大有助于让意面式的回调趋于平滑（富含 Ajax 调用的应用通常充斥着这些如意大利面条般纠缠在一起的回调）,而且也是因为Promise可以非常轻松地协调各种类型的异步任务。

运用 Promise 需要一些实践经验，特别是使用 `pipe` 时，不过这是非常值得培养的一种习惯。你可以在此窥见 JavaScript 的未来。返回 Promise 对象的 JavaScript API 越多，这些 API 就越有吸引力。

Microsoft 已经声明，Windows 8 的 Metro 环境将会提供一个基于 Promise 对象的 JavaScript API。[①]如果紧随潮流的开发者和 Microsoft 步调一致，整个世界势必会闻风景从。

---

① 参见 http://msdn.microsoft.com/en-us/library/windows/apps/br211867.aspx。

第 4 章

Async.js 的工作流控制

到目前为止，本书一直在讨论如何用抽象设计来管理遍布整个应用的异步任务。举例来说，PubSub 这种抽象设计允许应用程序把来源层的事件发布至其他层（譬如，在 MVC 三层架构设计模式中把从视图层的事件发布至模型层）。Promise 这种抽象设计允许将简单任务表示成对象，进而合并这些对象来表示更复杂的任务。总之，这些抽象设计一直在致力于帮助我们解决意面式回调这一问题。

不过我们的武器库仍然有一处短板：迭代。假设需要执行一组 I/O 操作（或者并行执行，或者串行执行），该怎么做呢？这个问题在 Node 中非常常见，以至于有了个专有名称：工作流控制（也称作控制工作流）。就像 Underscore.js 可以大幅度简化同步代码中的迭代一样，优秀的工作流控制库也可以消解异步代码中的套话。

目前最流行的工作流控制库当属 Caolan McMahon 开发的强大的 Async.js[①]。事实上，在我写作本书的时候，Async.js 是 npm[②]登记在案的请求第三多的库，它正与 Underscore.js、Express 这样的超级巨星一

① 参见 https://github.com/caolan/async。

② 参见 https://npmjs.org/。

起共沐荣光。

本章将探讨 Async.js 在 Node 环境下能做什么。（Async.js 也可运行于浏览器端，不过很少有客户端应用程序需要它。）还会简略介绍一个可代替 Async.js 的库——Tim Caswell 开发的超炫的 Step[①]。

Node及Async.js的安装

要想跟着本章照猫画虎，请从http://nodejs.org/获取最新版的Node。安装后，就可以运行npm工具（Node package manager，Node包管理器）了。使用下面这条命令即可安装Async.js及Step。

```
npm install -g async step
```

然后使用命令行`node file.js`即可运行任意的JavaScript文件（其中`file.js`为JavaScript文件名）。

4.1 异步工作流的次序问题

假设想先按字母顺序读取 `recipes`（菜谱）目录中的所有文件，接着把读取出的这些内容连接成一个字符串并显示出来。使用同步方法很容易做到这一点。

Asyncjs/synchronous.js

```
var fs = require('fs');
process.chdir('recipes'); // 改变工作目录

var concatenation = '';

fs.readdirSync('.')
  .filter(function(filename) {
```

① 参见 https://github.com/creationix/step。

```
    // 跳过不是文件的目录
    return fs.statSync(filename).isFile();
  })
    .forEach(function(filename) {
      // 内容添加到输出上
      concatenation += fs.readFileSync(filename, 'utf8')
    });

console.log(concatenation);
```

（注意，老式的 JavaScript 环境不支持使用 `forEach` 迭代器，如 IE6。使用一个像 Kris Kowal 之 es5-shim[①]这样的库可以解决这个问题。在第 6 章中，我们会学到如何只为有需求的浏览器提供库。）

不过，所有这种 I/O 阻塞的效率都极其低下，尤其是当应用程序还能同时做点其他事情的时候。问题在于不能单纯地将下面这行代码

```
concatenation += fs.readFileSync(filename, 'utf8');
```

换成异步代码：

```
fs.readFile(filename, 'utf8', function(err, contents) {
  if (err) throw err;
  concatenation += contents;
});
```

因为这么做根本无法保证按照做出 `readFile` 调用的次序来触发 `readFile` 调用的回调。`readFile` 仅仅负责告诉操作系统开始读取某个文件。对操作系统而言，读取短文件通常比读取长文件更快一些。因此，菜谱内容添加到 `concatenation` 字符串的次序是不可预知的。而且，在触发所有回调之后，必须要运行 `console.log`。

---

① 参见 https://github.com/kriskowal/es5-shim/。

要想使用很多异步任务并且希望结果可预知，需要先做一点规划。

4.2 异步的数据收集方法

我们先尝试在不借助任何工具函数的情况下来解决这个问题。笔者能想到的最简单的方法是：因前一个 `readFile` 的回调运行下一个 `readFile`，同时跟踪记录迄今已触发的回调次数，并最终显示输出。下面是笔者的实现结果。

Asyncjs/seriesByHand.js

```
var fs = require('fs');
process.chdir('recipes'); // 改变工作目录
var concatenation = '';

fs.readdir('.', function(err, filenames) {
  if (err) throw err;

  function readFileAt(i) {
    var filename = filenames[i];
    fs.stat(filename, function(err, stats) {
      if (err) throw err;
      if (! stats.isFile()) return readFileAt(i + 1);

      fs.readFile(filename, 'utf8', function(err, text) {
        if (err) throw err;
        concatenation += text;
        if (i + 1 === filenames.length) {
          // 所有文件均已读取，可显示输出
          return console.log(concatenation);
        }
        readFileAt(i + 1);
      });
    });
  }
```

```
  readFileAt(0);
});
```

如你所见，异步版本的代码要比同步版本多很多。如果使用 filter、forEach 这些同步方法，代码的行数大约只有一半，而且读起来也要容易得多。如果这些漂亮的迭代器存在异步版本该多好啊！使用 Async.js 就能做到这一点！

何时抛出亦无妨？

大家可能注意到了，在上面那个代码示例中笔者无视了自己在1.4节中提出的建议：从回调里抛出异常是一种糟糕的设计，尤其在成品环境中。不过，一个简单如斯的示例直接抛出异常则完全没有问题。如果真的遇到代码出错的意外情形，throw会关停代码并提供一个漂亮的堆栈轨迹来解释出错原因。

这里真正的不妥之处在于，同样的错误处理逻辑（即if(err) throw err）重复了多达3次！在4.2.2节，我们会看到Async.js如何帮助减少这种重复。

4.2.1 Async.js的函数式写法

我们想把同步迭代器所使用的 filter 和 forEach 方法替换成相应的异步方法。Async.js 给了我们两个选择。

- async.filter 和 async.forEach，它们会并行处理给定的数组。
- async.filterSeries 和 async.forEachSeries，它们会顺序处理给定的数组。

并行运行这些异步操作应该会更快，那为什么还要使用序列式方法呢？原因有两个。

- 前面提到的工作流次序不可预知的问题。我们确实可以先把结果存储成数组，然后再 `joining`（联接）数组来解决这个问题，但这毕竟多了一个步骤。
- Node 及其他任何应用进程能够同时读取的文件数量有一个上限。如果超过这个上限，操作系统就会报错。如果能顺序读取文件，则无需担心这一限制。

所以现在先搞明白 `async.forEachSeries` 再说。下面使用了 Async.js 的数据收集方法，直接改写了同步版本的代码实现。

Asyncjs/forEachSeries.js

```
var async = require('async');
var fs = require('fs');
process.chdir('recipes'); // 改变工作目录
var concatenation = '';

var dirContents = fs.readdirSync('.');

async.filter(dirContents, isFilename, function(filenames) {
  async.forEachSeries(filenames, readAndConcat, onComplete);
});

function isFilename(filename, callback) {
  fs.stat(filename, function(err, stats) {
    if (err) throw err;
    callback(stats.isFile());
  });
}

function readAndConcat(filename, callback) {
  fs.readFile(filename, 'utf8', function(err, fileContents) {
    if (err) return callback(err);
    concatenation += fileContents;
    callback();
  });
```

```
}

function onComplete(err) {
  if (err) throw err;
  console.log(concatenation);
}
```

现在我们的代码漂亮地分成了两个部分：任务概貌（表现形式为 async.filter 调用和 async.forEachSeries 调用）和实现细节（表现形式为两个迭代器函数和一个完工回调 onComplete）。

filter 和 forEach 并不是仅有的与标准函数式迭代方法相对应的 Async.js 工具函数。Async.js 还提供了以下方法：

- reject/rejectSeries，与 filter 刚好相反；
- map/mapSeries，1:1 变换；
- reduce/reduceRight，值的逐步变换；
- detect/detectSeries，找到筛选器匹配的值；
- sortBy，产生一个有序副本；
- some，测试是否至少有一个值符合给定标准；
- every，测试是否所有值均符合给定标准。

这些方法是 Async.js 的精髓，令你能够以最低的代码重复度来执行常见的迭代工作。在继续探索更高级的方法之前，我们先来看看这些方法的错误处理技术。

4.2.2　Async.js的错误处理技术

初始版本的异步代码实现一共有 3 条 throw 语句。到了 Async.js 版本只用了 2 条 throw，不过所有错误仍然会被抛出。Async.js 是怎么做到的呢？为什么不能只用 1 条 throw 呢？

简单来说，Async.js 遵守 Node 的约定。这意味着所有的 I/O 回调都形如 `callback(err, results...)`，唯一的例外是结果为布尔型的回调。布尔型回调的写法就是 `callback(result)`，所以上一代码示例中的 `isFilename` 迭代器需要自己亲自处理错误。

Asyncjs/forEachSeries.js

```
function isFilename(filename, callback) {
  fs.stat(filename, function(err, stats) {
    if (err) throw err;
    callback(stats.isFile());
  });
}
```

要怪就怪 Node 的 `fs.exists` 首开这一先河吧！而这也意味着使用了 Async.js 数据收集方法（`filter`/`filterSeries`、`reject`/`rejectSeries`、`detect`/`detectSeries`、`some`、`every` 等）的迭代器均无法报告错误。

对于非布尔型的所有 Async.js 迭代器，传递非 `null`/`undefined` 的值作为迭代器回调的首参数将会立即因该错误值而调用完工回调。这正是 `readAndConcat` 不用 `throw` 也能工作的原因。

Asyncjs/forEachSeries.js

```
function readAndConcat(filename, callback) {
  fs.readFile(filename, 'utf8', function(err, fileContents) {
    if (err) return callback(err);
    concatenation += fileContents;
    callback();
  });
}
```

所以，如果 `callback(err)`确实是在 `readAndConcat` 中被调用的，则这个 `err` 会传递给完工回调（即 `onComplete`）。Async.js 只负责保

证 onComplete 只被调用一次，而不管是因首次出错而调用，还是因成功完成所有操作而调用。

Asyncjs/forEachSeries.js

```
function onComplete(err) {
  if (err) throw err;
  console.log(concatenation);
}
```

Node 的错误处理约定对 Async.js 数据收集方法而言也许并不理想，但对于 Async.js 的 6 所有其他方法而言，遵守这些约定可以让错误干净利落地从各个任务流向完工回调。下一节会看到更多这样的例子。

4.3 Async.js 的任务组织技术

Async.js 的数据收集方法解决了一个异步函数如何运用于一个数据集的问题。但如果是一个函数集而不是一个数据集，又该怎么办呢？本节将探讨 Async.js 中一些可以派发异步函数并收集其结果的强大工具。

4.3.1 异步函数序列的运行

假设我们希望某一组异步函数能依次运行。在不使用工具函数的情况下，可能会编写出类似这样的代码：

```
funcs[0](function() {
  funcs[1](function() {
    funcs[2](onComplete);
  })
});
```

幸好我们还有 async.series 和 async.waterfall。这两个方法均接受一组函数（即任务列表）作为参数并按顺序运行它们，二者给任务列表中的每个函数均传递一个 Node 风格的回调。async.series 与 async. waterfall 之间的差别是，前者提供给各个任务的只有回调，而后者还会提供任务列表中前一任务的结果。（所谓“结果”，指的是各个任务传递给其回调的非错误的值。）

我们来看一个用延时实现的简单例子。

Asyncjs/seriesTimers.js

```
var async = require ('async');

var start = new Date;

async.series([
  function(callback) { setTimeout(callback, 100); },
  function(callback) { setTimeout(callback, 300); },
  function(callback) { setTimeout(callback, 200); }
], function(err, results) {
  // 显示自start而流逝的时间
  console.log('Completed in ' + (new Date -start) + 'ms');
});
```

（将 async.series 替换为 async.waterfall 不会对这个例子造成任何影响，因为这里各个任务的回调在运行时均不带参数。）

因为任务列表中的各个任务会按顺序完成，所以会在 600 毫秒之后（实际上比 600 毫秒稍长一些）运行完工回调（即因完成整个工作流事件而调用的回调，又称完工事件处理器）。Async.js 传递给任务列表中每个函数的回调好像在说：“出错了吗（回调的首参数是否为错误）？如果没出错，我就要收集结果（回调的次参数）并运行下一个任务了。”

下次再遇到一组需要按顺序运行的异步函数时，请试试 async.series 或 async.waterfall 吧。这二者中的一个很可能是最适合完成工作的工具。

4.3.2　异步函数的并行运行

Async.js 提供了 async.series 的并行版本，即 async.parallel。就像 async.series 一样，async.parallel 也接受一组形为 function(callback){...}的函数作为参数，但会再加上一个（可选的）在触发最末回调后运行的完工事件处理器。

前面那个延时例子重写如下。

Asyncjs/parallelTimers.js

```
var async = require ('async');
var start = new Date;
async.parallel([
  function(callback) { setTimeout(callback, 100); },
  function(callback) { setTimeout(callback, 300); },
  function(callback) { setTimeout(callback, 200); }
], function(err, results) {
  console.log('Completed in ' + (new Date -start) + 'ms');
});
```

async.series 完成工作流需要用掉 3 次延时的总和（约 600 毫秒），而 async.parallel 的用时只是最长的那次延时（约 300 毫秒）。

更为便利的是，Async.js 按照任务列表的次序向完工事件处理器传递结果，而不是按照生成这些结果的次序。这样，我们既拥有了并行机制的性能优势，又没有失去结果的可预知性。

async.series、async.waterfall、async.parallel 与那些数据

收集方法一起，诠释了 Async.js 的内核与灵魂：为最常见的异步情景提供简单又省时的工具函数。

4.4 异步工作流的动态排队技术

大多数情况下，前两节介绍的那些简单方法足以解决我们的异步窘境，但 `async.series` 和 `async.parallel` 均存在各自的局限性。

- 任务列表是静态的。一旦调用了 `async.series` 或 `async.parallel`，就再也不能增减任务了。
- 不可能问："已经完成多少任务了？"任务处于黑箱状态，除非我们自行从任务内部派发更新信息。
- 只有两个选择，要么是完全没有并发性，要么是不受限制的并发性。这对文件 I/O 任务可是个大问题。如果要操作上千个文件，当然不想因按顺序操作而效率低下，但如果试着并行执行所有操作，又很可能会激怒操作系统。

Async.js 提供了一种可以解决上述所有问题的全能方法：`async.queue`。

4.4.1 深入理解队列

`async.queue` 的底层基本理念令人想起 DMV（Dynamic Management View，动态管理视图）。它可以同时应对很多人（最多时等于在岗办事员的数目），但并不是每位办事员前面各排一个队，而是维持着一个排号队列。人到了就排队，并取得一个排队号码。任何一个办事员空闲时，就会叫下一个排队号码。

`async.queue` 的接口比 `async.series` 和 `async.parallel` 稍微复

杂一些。`async.queue` 接受的参数有两个：一个是 worker（办事员）函数，而不是一个函数列表；一个是代表着 concurrency（并发度）的值，代表了办事员最多可同时处理的任务数。`async.queue` 的返回值是一个队列，我们可以向这个队列推入任意的任务数据及可选的回调。

下面是一个小例子。

Asyncjs/simpleQueue.js

```
var async = require('async');

function worker(data, callback) {
  console.log(data);
  callback();
}
var concurrency = 2;
var queue = async.queue(worker, concurrency);
queue.push(1);
queue.push(2);
queue.push(3);
```

不论并发度是多少（只要不小于 1），都会得到以下输出。

```
1
2
3
```

不过内在还是有点小区别：并发度为 2 时，需要两轮才能遍历事件队列；如果并发度为 1，则需要 3 轮才能遍历，每轮输出一行代码；如果并发度为 3 或更大的值，则只需要 1 轮即可遍历。

并发度为 0 的队列不会做任何事情。如果想要最大的并发度，请直接使用 `Infinity` 关键字。

4.4.2　任务的入列

虽然 queue.push 与[].push 同名，但二者存在两个很关键的差别。

第一个差别。请看这行代码：

```
queue.push([1, 2, 3]);
```

它等价于下面这 3 行代码：

```
queue.push(1);
queue.push(2);
queue.push(3);
```

这意味着不能直接使用数组作为任务的数据。不过可以使用其他任何东西（甚至函数）作为任务的数据。事实上，如果想让 async.queue 像 async.series/async.parallel 那样也使用一组函数作为任务列表，只需定义一个其次参数会直接传递给其首参数的 worker 函数。

```
function worker(task, callback) {
  task(callback);
}
var concurrency = 2;
var queue = async.queue(worker, concurrency);
queue.push(tasks);
```

第二个差别。async.queue 中的每次 push 调用可附带提供一个回调函数。如果提供了，该回调函数会直接送给 worker 函数作为其回调参数。因此，（假设 worker 函数确实运行了其回调，即它未因抛出错误而直接关停）下面这个例子将会触发 3 次输出事件，即输出 3 次 'Task complete!'。

```
queue.push([1, 2, 3], function(err, result) {
  console.log('Task complete!');
});
```

对 async.queue 而言，push 方法的回调函数非常重要，因为 async.queue 不像 async.series/async.parallel 那样可以在内部存储每次任务的结果。如果想要这些结果，就必须自行去捕获。

4.4.3　完工事件的处理

和 async.series 及其类似方法一样，我们也可以给 async.queue 指定一个完工事件处理器。不过，这时并不是传递完工事件处理器作为 async.queue 方法的参数，而是要附加它作为 async.queue 对象的 drain（排空）属性。（请想象有一个盆，里面满是待完成的任务。当最后一个任务也排空流出盆时，就会触发完工事件的回调。）下面用计时器做了一个示例。

Asyncjs/queueTimers.js

```
var async = require('async');

function worker(data, callback) {
  setTimeout(callback, data);
}
var concurrency = 2;
var queue = async.queue(worker, concurrency);
var start = new Date;
queue.drain = function() {
  console.log('Completed in ' + (new Date -start) + 'ms');
};

queue.push([100, 300, 200]);
```

回想一下：async.series 完成工作流需要大约 600 毫秒的时间（3 次延时的总和），而 async.parallel 只用掉约 300 毫秒（即最长的那次延时）。这里的并发度为 2，所以工作流一开始就会并行运行前两次延时。不过结束运行那次 100 毫秒的延时之后，队列里的下一个

任务（即 200 毫秒的延时）将会立即开始运行。因此，在这种情况下，async.queue 和 async.parallel 差不多在同一时刻结束运行。这里工作流的次序起到了关键作用：如果第 3 次入列的是那个 300 毫秒的延时任务，则整个队列需用时约 400 毫秒才能完成。

注意，可以一直调用 push 方法向队列推入更多后续任务，而且，每当队列中的末任务结束运行时，都会触发一次 drain（也就是说，如果任务队列完工之后再让新任务入列，新任务队列的完工也同样会触发完工事件处理器）。遗憾的是，这意味着 async.queue 不能像 async.waterfall 那样提供清晰排序的结果。如果想收集那些入列的任务的结果数据，就只能靠自己了。

4.4.4　队列的高级回调方法

尽管 drain 常常是我们唯一要用到的事件处理器，但 async.queue 还是提供了其他一些事件及其处理器。

- 队末任务开始运行时，会调用队列的 empty 方法。（队末任务运行结束时，会调用队列的 drain 方法。）
- 达到并发度的上限时，会调用队列的 saturated 方法。
- 如果提供了一个函数作为 push 方法的次参数，则在结束运行给定任务时会调用该函数，或在给定任务列表中的每个任务结束运行时均调用一次该函数。

在本节中，我们已经看到 async.queue 为何成为了最强大的 Async.js 函数之一。如果需要在有限的并发度下运行大量的异步任务，请考虑使用 async.queue。

Async.js 是使用最广泛、当然也是功能最丰富的 JavaScript 工作流控

制库，针对它的讨论到此为止。不过，请勿以为对于所有基于回调驱动的任务来说，Async.js 都是一种恰当的工具。下面将介绍 Async.js 最大的竞争对手之一——Step。

4.5　极简主义者 Step 的工作流控制

Tim Caswell 的 Step[①]是一个轻量级的 JavaScript 库。事实上，Step 的 API 集只有一个函数：`Step`。

`Step` 接受一个函数列表作为参数，例子如下。

```
Step(task1, task2, task3);
```

其中的每一个函数都有 3 种控制工作流的方式。

- 它可以调用 `this`，让 Step 直接运行列表中的下一个函数。
- 它可以调用 *n* 次由 `this.parallel` 或 `this.group` 生成的回调，以告诉 Step 应运行列表中的下一个函数 *n* 次。
- 它可以返回一个值，这也会让 Step 运行列表中的下一个函数。这简化了同步函数与异步函数的混合使用。

下一个函数会收到上一个函数的结果（或者是其返回的值，或者是其传递给 `this` 的参数），或者是上一函数所有实例的结果（如果因 `this.parallel` 或 `this.group` 而运行该函数的话）。其区别在于，`this.parallel` 会将结果作为一个个单独的参数来提供，而 `this.group` 会将结果合并成数组。

---

① 参见 https://github.com/creationix/step。

在笔者写作本书时，整个 Step 库的代码仅有 152 行（含注释），但它足以应对大多数异步工作流。这种极简主义的不足是，要想理解 Step 生成的工作流，只有去通读其中的每一个函数。而无所不包、无所不能的 Async.js 所生成的工作流一般更能够自我解释。

不过，如果想自己动手、亲历亲为，那么用 Step 来写一些类似于 Async.js 的工具函数会是一种很好的练习。例如，下面只用 11 行代码就实现了等价的 `async.map`。

Asyncjs/stepMap.js

```
var Step = require('step');

function stepMap(arr, iterator, callback) {
  Step(
    function() {
      var group = this.group();
      for (var i = 0; i < arr.length; i++) {
        iterator(arr[i], group());
      }
    },
    callback
  );
}
```

笔者认为使用 Step 能让人耳目一新。使用 Async.js 主要是想找到最适合任务的那个工具函数，而 Step 却能鼓励我们透彻地思考问题并编写出优雅高效的解决方案。

4.6　小结

本章介绍了如何借助合适的工作流控制函数用最少量的重复代码来实现常见的异步工作流模式。Async.js 已经成为首屈一指的工作流控

制库，它既提供了健壮的迭代式数据收集方法，又实现了可靠的调度任务的方法。如果遇到了工作流控制问题，Async.js 很可能有解决方案。如果你喜欢自己解决，不妨使用 Step。

Isaac Schlueter 是 Node.js 项目的首席开发者，他曾经做过一个非常小的工作流控制库。该库名为 Slide①，可用于 npm 环境中。在 Slide 的 README（请先阅读）文件中，Isaac Schlueter 写道：

> 应该把 Slide 当作一个示例，它演示了如何编写属于自己的工作流控制工具。如果没有亲手写过工作流控制库，就永远不会真正地了解它。

笔者觉得这种说法并不对。编写工作流控制库确实是很好的练习，但没有必要为了看看轮子是怎么工作的就重新发明轮子。随着 JavaScript 生态系统的成熟，工作流控制的概念会越来越普及，越来越标准化。暂时而言，如果你的应用需要工作流控制，那么最重要的是选择一个好的工作流控制库并掌握它。

① 参见 https://github.com/isaacs/slide-flow-control。

第 5 章

worker 对象的多线程技术

本书一开始就说过事件是多线程技术的替代品。更准确地说，事件能够代替一种特殊的多线程，即应用程序进程可拆分成多个部分同时运行的多线程技术（或者通过中断技术虚拟实现，或者通过多个 CPU 内核真正实现）。但如果不同线程中运行的代码需要访问同一数据，则会出现问题。即便是一行简单如斯的代码

```
i++;
```

一旦允许不同的线程同时修改同一个 `i`，也会变成毁灭性的海森堡蚁虫（Heisenbug）[①]之家。幸亏这种多线程技术在 JavaScript 里是不可能的。

另一方面，向多颗 CPU 内核分发任务又日益成为基本需求，因为这些 CPU 内核不再像过去期望的那样持续地大幅提升效率。因此，我们需要多线程技术。那这是否意味着要放弃基于事件的编程呢?

① 参见 http://en.wikipedia.org/wiki/Heisenbug。Heisenbug 指的是在成品环境下会不经意出现，但费尽九牛二虎之力却无法重现的计算机 bug。——译者注

恰恰相反！尽管只运行在一个线程上确实不怎么理想，但天真地将应用直接分发给多个内核更加糟糕。多内核系统的那些内核如果必须经常交谈以避免相互拆台，那么整个系统就会慢得像蜗牛爬一样。最好能让每颗内核领取一项独立的作业，然后偶尔同步一下。

JavaScript 中的 worker（办事员）对象就是这么干的。应用程序的主线程可以这样跟 worker 说："去，开一个单独的线程来运行这段代码。"worker 可以给我们发送消息（反之亦可），其表现形式是事件队列中运行的回调（还能是别的吗？）。简而言之，与不同线程进行交互的方式与在 JavaScript 中进行 I/O 操作一模一样。

本章会讲述浏览器端及 Node 环境中的 worker 对象，还会讨论一些实践应用。

线程与进程的对比

本章反复絮叨的线程/进程这两个词是可以互换的。但在操作系统的层面，这两者存在一个重要的区别：同一个进程内的多个线程之间可以分享状态，而彼此独立的进程之间则不能。但在JavaScript环境中，由worker对象运行的并发代码从来不会分享状态。所以，尽管可以使用轻量级OS线程来实现worker对象，但它们的行为更像是进程。

有一些Node库为了效率而允许破坏关于状态分享的游戏规则，其中最著名的当属Threads-A-GoGo[a]。这些已经超出了本章的范畴，本章只关心标准JavaScript环境中的并发性。

a. 参见https://github.com/xk/node-threads-a-gogo。

5.1　网页版 worker 对象

网页版 worker 对象是大名鼎鼎的 HTML5 标准的一部分。要想生成 worker 对象，只需以脚本 URL 为参数来调用全局 `Worker` 构造函数即可。

```
var worker = new Worker('worker.js');
worker.addEventListener('message', function(e) {
  console.log(e.data); // 重复由 postMessage 发送的任何东西
});
```

（通常，我们只用到 `message` 事件的 `data` 属性。如果已经把同一个事件处理器绑定至多个 worker 对象，则可能还要用到 `e.target` 以定位是哪一个 worker 对象触发了事件。）

现在我们知道如何监听 worker 对象了。worker 对象的交互接口呈现出便利的对称设计：我们可以使用 `worker.postMessage` 来发送消息，而 worker 本身可以使用 `self.addEventListener('message',...)` 来接收消息。下面是一个完整的例子。

```
// 主脚本
var worker = new Worker('boknows.js');
worker.addEventListener('message', function(e) {
  console.log(e.data);
});
worker.postMessage('football');
worker.postMessage('baseball');

// boknows.js
self.addEventListener('message', function(e) {
  self.postMessage('Bo knows ' + e.data);
});
```

大家可以在笔者创建的一个小网站上（Web Worker Sandbox[①]）“把玩”worker 对象的这种消息传递接口。创建一个新例子之后，你会得到一个可与他人分享的唯一 URL 地址。

5.1.1　网页版worker对象的局限性

网页版 worker 对象的首要目标是，在不损害 DOM 响应能力的前提下处理复杂的计算。它有以下几种潜在用法：

- 解码视频。流入的视频采用 Broadway 实现的 H.264 编解码器[②]。
- 采用斯坦福的 JavaScript 加密库[③]加密通信。
- 解析网页式编辑器中的文本。没错，就是 Ace 编辑器[④]。

事实上，Ace 默认已经这么做了。在基于 Ace 的编辑器中键入代码时，Ace 需要先进行一些相当繁重的字符串分析，然后再使用恰当的语法高亮格式更新 DOM。在现代浏览器中，这种分析工作通常会另开一个独立线程来做，以确保编辑器能保持平滑度和响应度。

通常情况下，worker 对象会把自己的计算结果发送给主线程，由主线程去更新页面。为什么不直接更新页面呢？这主要是为了保护 JavaScript 异步抽象概念，使其免受影响。如果 worker 对象可以改变页面的标记语言，那么最终的下场就会和 Java 一样——必须将 DOM 操控代码封装成互斥量和信号量以避免竞态条件。

基于类似的理由，worker 对象也看不到全局的 `window` 对象和主线程

① 参见 http://webworkersandbox.com/。
② 参见 https://github.com/mbebenita/Broadway。
③ 参见 http://crypto.stanford.edu/sjcl/。
④ 参见 http://ace.ajax.org/。

及其他 worker 线程中的其他任何对象。通过 `postMessage` 发送的对象会透明地作序列化和反序列化，想想 `JSON.parse(JSON.stringify(obj))`吧。因此，原始对象的变化不会影响该对象在其他线程中的副本。

worker 对象甚至不能使用老实可靠的 `console` 对象。它只能看到自己的全局对象 `self`，以及 `self` 已捆绑的所有东西：标准的 JavaScript 对象（如 `setTimeout` 和 `Math`），以及浏览器的 Ajax 方法。

对了，还有 Ajax！worker 对象可以随意使用 `XMLHttpRequest`。如果浏览器端支持的话，它甚至能使用 `WebSocket`。这意味着 worker 可以直接从服务器拉取数据。如果要处理大量数据（譬如说，流传输一段需要解码的视频），相比于使用 `postMessage` 来序列化这些数据，将数据封闭在一个线程中会更好。

还有一个特殊的 `importScripts` 函数可以同步地加载并运行指定的脚本。

```
importScripts('https://raw.github.com/gist/1962739/danika.js');
```

正常情况下，同步加载是绝对禁忌，不过 worker 对象跑的是另一个线程。既然 worker 已经无所事事了，那么阻塞就阻塞好啦！

5.1.2 支持网页版worker的浏览器

在桌面端，Chrome、Firefox 及 Safari 都已实现网页版 worker 标准好几年了，现在 IE 10 也内嵌了这一标准。移动端的支持情况也可圈可点。最新的 iOS 版 Safari 支持网页版 worker，但最新版的 Android 浏览器还不支持。在笔者写作本书的时候，Caniuse.com 统计认为浏览

器端对网页版 worker 的支持率达 59.12%。[①]

简单来说，大家不能指望自己网站的用户支持网页版 worker。不过，即便 `window.Worker` 不可用，也可以轻松用垫片技术保证目标脚本的正常运行。毕竟，网页版 worker 只是一种性能增强工具。

请认真地对多个浏览器端测试网页版 worker，因为不同的浏览器实现之间存在一些至关重要的差异。例如，Firefox 允许 worker 对象孵化出自己的"worker 子对象"，但 Chrome 目前还不支持这么做。

5.2 cluster 带来的 Node 版 worker

在 Node 的早期阶段，有很多 API 竞相抢食多线程这块肥肉，其中大多数 API 的实现都很蠢笨，不断要求用户搞出多个服务器实例以监听不同的 TCP 端口，然后再通过代理回钩到真正的端口。直到发行了 0.6 版本，才推出了一个支持多个进程绑定至同一端口的标准 API：`cluster`（群集）[②]。

通常情况下，会为了追求最佳性能而使用 `cluster` 按每颗 CPU 内核分化出一个进程（尽管每个进程是否能真正得到属于自己的内核仍完全取决于底层的操作系统）。

Multithreading/cluster.js

```
var cluster = require('cluster');
if (cluster.isMaster) {
  // 分化出 worker 对象
  var coreCount = require('os').cpus().length;
```

① 参见 http://caniuse.com/webworkers。
② 参见 http://nodejs.org/docs/latest/api/cluster.html。

```
  for (var i = 0; i < coreCount; i++) {
    cluster.fork();
  }
  // 绑定 death 事件
  cluster.on('death', function(worker) {
    console.log('Worker ' + worker.pid + ' has died');
  });
} else {
  // 立即死去
  process.exit();
}
```

输出形如：

```
Worker 15330 has died
Worker 15332 has died
Worker 15329 has died
Worker 15331 has died
```

其中每行输出对应一颗 CPU 内核。

这段代码看似莫名其妙，其玄妙之处在于，网页版 worker 对象会加载一个独立的脚本，而 Node 版 worker 对象则由 `cluster.fork()`把运行自己的同一个脚本再次加载成一个独立的进程。运行中的脚本要想知道自己是主进程还是 worker 对象，唯一的办法就是检查 `cluster.isMaster`。

为什么做出这样的设计决策呢？因为 Node 版多线程技术的使用情况与浏览器端有很大不同。浏览器可以将任意多余的线程降格为后台任务，而 Node 服务器则要留出计算资源以保障其主要任务：处理请求。

（使用 `child_process.fork`[①]也可以将外部脚本加载为独立的进程

① 参见 http://nodejs.org/docs/latest/api/child _ process.html。

来运行。除了子进程不能分享 TCP 端口之外，child_process.fork 的功能几乎和 cluster.fork 完全一样，事实上 cluster 内在里就使用了 child_process。）

5.2.1　Node版worker的交互接口

和网页版 worker 对象一样，cluster 分化出的 worker 对象通过发送消息事件来与主进程交流，反之亦然。不过，这里的 API 稍有不同。

Multithreading/clusterMessage.js

```
var cluster = require('cluster');
if (cluster.isMaster) {
  // 分化 worker 对象
  var coreCount = require('os').cpus().length;
  for (var i = 0; i < coreCount; i++) {
    var worker = cluster.fork();
    worker.send('Hello, Worker!');
    worker.on('message', function(message) {
      if (message._queryId) return;
      console.log(message);
    });
  }
} else {
  process.send('Hello, main process!');
  process.on('message', function(message) {
    console.log(message);
  });
}
```

输出形如：

```
Hello, main process!
Hello, main process!
Hello, Worker!
Hello, Worker!
Hello, main process!
```

```
Hello, Worker!
Hello, main process!
Hello, Worker!
```

这里的输出次序是不可预知的，因为每个线程都竞相抢占 `console.log`。（大家必须用 Ctrl+C 快捷键手动结束此进程。）

和网页版 worker 一样，Node 版 worker 对象的 API 也是对称的，此端的 `send` 调用会触发彼端的`'message'`事件。但需要注意的是，`send` 的参数（更确切的说法是参数的序列化副本）直接由`'message'`事件指定，而不是附加作为事件的 `data` 属性。

注意到主消息处理器的这行代码了吗？

```
if (message._queryId) return;
```

Node 有时会基于 worker 对象发送自己的消息，发送命令始终形如：

```
{ cmd: 'online', _queryId: 1, _workerId: 1 }
```

忽略这些内部消息也没什么危险，但要知道这些消息会在后台施展一些重要的魔法。其中最著名的魔法是：多个 worker 对象试图监听（`listen`）一个 TCP 端口时，Node 利用内部消息来允许分享该端口。

5.2.2 Node版worker对象的局限性

大多数情况下，`cluster` 对象和网页版 worker 对象遵守同样的规则：有一个主线程和多个 worker 线程，它们之间的交流基于一些带有序列化对象或附连字符串的事件。不过，网页版 worker 在浏览器端显然是二等公民，而 Node 版 worker 却几乎拥有主线程的所有权利和权限，但以下这些除外（但不限于这些）：

- 关停应用程序的能力；
- 孵化出更多 worker 对象的能力；
- 彼此交流的能力。

这使得主线程必须承担起作为所有线程间通信之中转中心的重任。幸运的是，这种不便可以通过像 Roly Fentanes 之 Clusterhub[①]这样的库抽象出去。

在本节中，我们看到了 worker 对象为何成为 Node 的一个有机组成部分，以及它允许服务器充分利用多颗内核而无需运行多个应用实例。Node API 之 `cluster` 支持并发运行同一个脚本（一个主进程和任意多个 worker 进程）。为了尽可能减少线程间通信的开销，线程间分享的状态应该存储在像 Redis 这样的外部数据库中。

5.3 小结

尽管说这话为时尚早，但笔者依然认为多内核 JavaScript 的前途一片光明。任何应用只要可以拆分成大体独立且彼此间只需定期交互的多个进程，那么 worker 对象就是一种可以充分利用 CPU 性能的致胜方案。分布式计算从来都是妙趣横生的。

① 参见 https://github.com/fent/clusterhub。

第 6 章

异步的脚本加载

先来看这行代码：

```
<script src = "allMyClientSideCode.js"></script>
```

这有点儿……不怎么样。“这该放在哪儿？”开发人员会奇怪，“靠上点，放到<head>标签里？还是靠下点，放到<body>标签里？”这两种做法都会让富脚本站点的下场很凄惨。<head>标签里的大脚本会滞压所有页面渲染工作，使得用户在脚本加载完毕之前一直处于“白屏死机”[①]状态。而<body>标签末尾的大脚本只会让用户看到毫无生命力的静态页面，原本应该进行客户端渲染的地方却散布着不起作用的控件和空空如也的方框。

完美解决这个问题需要对脚本分而治之：那些负责让页面更好看、更好用的脚本应该立即加载，而那些可以待会儿再加载的脚本稍后再加载。但是怎样才能既滞压这些脚本，又能保证它们在被调用时的可用性呢？

① *JavaScript Performance Rocks!*一书的作者 Amy Hoy 和 Thomas Fuchs 杜撰了“白屏死机”（White Screen of Death）这个词。这本大作可视为目前前端 JavaScript 性能优化领域最权威的资料，参见 http://javascriptrocks.com/performance/。——译者注

志在解决这一问题的一些技术在过去几年里已逐渐普及开来。本章将介绍 HTML5 之 `async`/`defer` 属性的作用，以及两个流行的脚本加载库：yepnope 和 Require.js。

> **Node.js的异步加载技术**
>
> 由于Node异步模块加载技术的用处微乎其微，本章将紧紧围绕浏览器端的异步加载问题。Node一度提供过异步版本的`require`，但到0.3版本就删除掉了。如果有兴趣了解为什么，请浏览 https://groups.google.com/d/msg /nodejs/y_- LZqltb1A/mmpYLlLurqkJ。

6.1　局限性与补充说明

在切入主题之前，敬请注意以下重要事宜：

- 本章中的技术不适用于内联脚本，即那些在页面标记中直接定义的脚本。请尽量避免使用内联技术。如果非得内联脚本，请勿试图对该脚本使用 `defer`/`async` 属性。
- 使用本章任何技术时请勿使用 `document.write`。异步加载脚本中的 `document.write` 会表现出不可预知的行为。最好大家根本就不知道 `document.write` 是什么。在此，只要知道 `document.write` 相当于操控 DOM 时的 `GOTO` 语句就行了。
- 这份“指南”不够严谨。为简洁起见，笔者忽略了一些特定于平台的重要细节。例如，某些移动浏览器会拒绝缓存超过一定尺寸的大脚本。因此，如果目标是这些设备的话，保持脚本的小巧就很重要了。

页面加载优化技术是一个相关专著已汗牛充栋的丰富主题，而脚本加载技术不过是其中的一小部分。但对那些不曾使用过异步加载技术的富脚本站点来说，使用本章的一些技术能唾手可得丰厚的回报。

6.2 <script>标签的再认识

虽然有些夸大其词，但笔者仍然想说`<script>`一直以来都是最重要的 HTML 标签。不信？请想象一下吧，有个网页虽然只有一个`<script>`标签却能做任何事情！`<script>`标签能够空手套白狼，凭空变出文档，加载任何自己想要的资源。相比之下，没有`<script>`标签的页面即使再炫目也存在明显的缺陷，即无法响应用户的动作，最复杂的回应也不过是肤浅的 CSS 变换。

现代浏览器中的`<script>`标签分成了两种新类型：经典型和非阻塞型。本节将讨论如何运用这两种标签来尽快加载页面。

6.2.1 阻塞型脚本何去何从

标准版本的`<script>`标签常常被称作*阻塞型*标签。这个词必须放在上下文中进行理解：现代浏览器看到阻塞型`<script>`标签时，会跳过阻塞点继续读取文档及下载其他资源（脚本和样式表）。但直到脚本下载完毕并运行之后，浏览器才会评估阻塞点之后的那些资源。

因此，如果网页文档的`<head>`标签里有 5 个阻塞型`<script>`标签，则在所有这 5 个脚本均下载完毕并运行之前，用户除了页面标题之外看不到任何东西。不仅如此，即便这些脚本运行了，它们也只能看到阻塞点之前的那部分文档。如果想看到`<body>`标签中正等待加载的那些

好东西，就必须给像 document.onreadystatechange 这样的事件绑定一个事件处理器。

基于上述原因，现在越来越流行把脚本放在页面<body>标签的尾部。这样，一方面用户可以更快地看到页面，另一方面脚本也可以主动亲密接触 DOM 而无需等待事件来触发自己。对大多数脚本而言，这次“搬家”是个巨大的进步。

但并非所有脚本都一样。在向下搬动脚本之前，请先问自己 3 个问题。

- 该脚本是否有可能被<body>标签里的内联 JavaScript 直接调用？答案可能一目了然，但仍值得核查一遍。
- 该脚本是否支持老式浏览器识别 HTML5 元素？Modernizr[①]支持，也正因为这个原因，最佳实践项目的典范 HTML5 Boilerplate[②]直接在文档顶部包含了 Modernizr。
- 该脚本是否会影响已渲染页面的外观？Typekit 宿主字体就是一个例子。如果把 Typekit 脚本放在文档末尾，那么页面文本就会渲染两次，即读取文档时即刻渲染，脚本运行时再次渲染。

上述问题只要有一个答案是肯定的，那么该脚本就应该放在<head>标签中，否则就应该放在<body>标签中。抽象、集中这两类脚本之后会让文档形如：

```
<html>
<head>
  <!--metadata and stylesheets go here -->
  <script src="headScripts.js"></scripts>
</head>
```

① 参见 http://modernizr.com/。
② 参见 http://html5boilerplate.com/。

```
<body>
  <!-- content goes here -->
  <script src="bodyScripts.js"></script>
</body>
</html>
```

这确实大大缩短了加载时间，但要注意一点，这可能让用户有机会在加载 bodyScripts.js 之前与页面交互。

6.2.2 脚本的延迟运行

上一节中建议将大多数脚本放在<body>中，因为这样既能让用户更快地看到网页，又能避免操控 DOM 之前绑定“就绪”事件的开销。但这种方式也有一个缺点，即浏览器在加载完整个文档之前无法加载这些脚本，这对那些通过慢速连接传送的大型文档来说会是一大瓶颈。

理想情况下，脚本的加载应该与文档的加载同时进行，并且不影响 DOM 的渲染。这样，一旦文档就绪就可以运行脚本，因为已经按照<script>标签的次序加载了相应脚本。

如果大家已经读到这里了，那么一定会迫不及待地想写一个自定义 Ajax 脚本加载器以满足这样的需求！不过，大多数浏览器都支持一个更为简单的解决方案。

```
<script defer src = "deferredScript.js">
```

添加 defer（延迟）属性相当于对浏览器说：“请马上开始加载这个脚本吧，但是，请等到文档就绪且所有此前具有 defer 属性的脚本都结束运行之后再运行它。”在文档<head>标签里放入延迟脚本，既能带来脚本置于<body>标签时的全部好处，又能让大文档的加载速度大幅提升！

有什么不足？并非所有浏览器都支持 `defer`。在笔者写作本书的时候，甚至最新版的 Opera 都忽略了这个属性。[①]这意味着，如果想确保自己的延迟脚本能在文档加载后运行，就必须将所有延迟脚本的代码都封装在诸如 jQuery 之`$(document).ready`之类的结构中。这是值得的，因为差不多 97%的访客都能享受到并行加载的好处，同时另外 3%的访客仍然能使用功能完整的 JavaScript。

使用 `defer` 属性把 bodyScript.js 换成 deferredScript.js，于是上一节的页面例子改进如下：

```
<html>
<head>
  <!-- metadata and stylesheets go here -->
  <script src="headScripts.js"></scripts>
  <script defer src="deferredScripts.js"></script>
</head>
<body>
  <!-- content goes here -->
</body>
</html>
```

请记住 deferredScripts 的封装很重要，这样即使浏览器不支持 `defer`，deferredScripts 也会在文档就绪事件之后才运行。如果页面主体内容远远超过几千字节，那么付出这点代价是完全值得的。

6.2.3　脚本的完全并行化

如果你是斤斤计较到毫秒级页面加载时间的完美主义者，那么 `defer` 也许就像是淡而无味的薄盐酱油。你可不想一直等到此前所有的 `defer` 脚本都运行结束，当然也肯定不想等到文档就绪之后才运行这

① 参见 http://caniuse.com/#search=defer。

些脚本，更别提为了照顾 Opera 还使用什么`$(document).ready`了。你就是想尽快加载并且尽快运行这些脚本。

这也正是现代浏览器提供了 `async`（异步）属性的原因。

```
<script async src = "speedyGonzales.js">
<script async src = "roadRunner.js">
```

如果说 `defer` 让我们想到一种静静等待文档加载的有序排队场景，那么 `async` 就会让我们想到混乱的无政府状态。前面给出的那两个脚本会以任意次序运行，而且只要 JavaScript 引擎可用就会立即运行，而不论文档就绪与否。因此，抛开对速度的渴求，我们有什么理由用 `async` 呢？

对大多数脚本来说，`async` 是一块难以下咽的鸡肋。`async` 不像 `defer` 那样得到广泛的支持，因此很少有用户会注意到性能的提升。[①]同时，由于异步脚本会在任意时刻运行，它实在太容易引起海森堡蚁虫之灾了（脚本刚好结束加载时就会蚁虫四起）。

但是，对那些一心一意搞独立的脚本来说，`async` 确实是一次不大但很重要的胜利。获取一个负责添加反馈小部件的第三方脚本，再获取一个负责添加技术支持聊天框的第三方脚本，怎么样？页面没有它们也运行得很好，而且也不在乎它们谁先运行谁后运行。因此，对这些第三方脚本使用 `async` 属性，相当于一分钱没花就提升了它们的运行速度。

① 参见 http://caniuse.com/#search=async。

Async+Defer=?

大家可能会问："如果我对同一个脚本既用defer又用async，会怎么样呢？"答案是，在那些同时支持这两个属性的浏览器中，async会覆盖掉defer。由于defer有着更广泛的支持，而且具有async的主要优势（允许在下载脚本的同时进行DOM的渲染），因此我们建议尽量使用defer代替async。

上一个页面示例再添加两个独立的第三方小部件，得到的结果如下：

```
<html>
<head>
  <!-- metadata and stylesheets go here -->
  <script src="headScripts.js"></scripts>
  <script src="deferredScripts.js" defer></script>
</head>
<body>
  <!-- content goes here -->
  <script async defer src="feedbackWidget.js"></script>
  <script async defer src="chatWidget.js"></script>
</body>
</html>
```

这个页面结构清晰展示了脚本的优先次序。对于绝大多数浏览器，DOM的渲染只会延迟至 headScripts.js 结束运行时。进行 DOM 渲染的同时会在后台加载 deferredScripts.js。接着，在 DOM 渲染结束时将运行 deferredScripts.js 和那两个小部件脚本。这两个小部件脚本在那些支持 async 的浏览器中会做无序运行。如果不确定这是否妥当，请勿使用 async！

本节讨论了<script>标签的位置和属性如何对页面用户等待时间造成了巨大的差异。下一节将学习如何使用脚本来加载其他脚本，以进一步利用异步加载原理。

6.3 可编程的脚本加载

虽然<script>标签简单得令人心动，但有些情况确实需要更精致的脚本加载方式。我们可能只想给那些满足一定条件的用户加载某个脚本，譬如白金会员或达到一定级别的玩家，也可能只想当用户单击激活时才加载某个特性，譬如聊天小部件。

本节将讨论如何用脚本加载其他脚本。简要介绍一些低级方法后，将讨论两个会让脚本加载易如反掌的流行库：yepnope 和 Require.js。

6.3.1 直接加载脚本

在浏览器 API 层面，有两种合理的方法来抓取并运行服务器脚本。

- 生成 Ajax 请求并用 `eval` 函数处理响应。
- 向 DOM 插入<script>标签。

后一种方法更好，因为浏览器会替我们操心生成 HTTP 请求这样的事。再者，`eval` 也有一些实际问题：泄漏作用域，调试搞得一团糟，而且还可能降低性能。因此，要想加载名为 feature.js 的脚本，我们应该用类似下面这样的代码来插入<script>标签。

```
var head = document.getElementsByTagName('head')[0];
var script = document.createElement('script');
script.src = '/js/feature.js';
head.appendChild(script);
```

稍等，我们如何才能知道脚本何时加载结束呢？我们可以给脚本本身添加一些代码以触发事件，但如果要为每个待加载脚本都添加这样的代码，那也太闹心了。或者是另外一种情况，即我们不可能给第三方服务器上的脚本添加这样的代码。HTML5 规范定义了一个可以绑定

回调的 onload 属性。

```
script.onload = function() {
  // 现在可以调用脚本里定义的函数了
};
```

不过，IE8 及更老的版本并不支持 onload，它们支持的是 onreadystatechange。某些浏览器在插入<script>标签时还会出现一些“灵异事件”。而且，这里甚至还没谈到错误处理呢！为了避免所有这些令人头疼的问题，笔者在此强烈建议使用脚本加载库。

6.3.2　yepnope的条件加载

yepnope[①]是一个简单的、轻量级的脚本加载库（压缩后的精简版只有 1.7KB），其设计目标就是真诚服务于最常见的动态脚本加载需求。yepnope 可以独立使用，也可以作为 Modernizr 特性检测库的一部分。

yepnope 最简单的用法是，加载脚本并对脚本完成运行这一事件返回一个回调。

```
yepnope({
  load: 'oompaLoompas.js',
  callback: function() {
    console.log('Oompa-Loompas ready!');
  }
});
```

还是无动于衷？下面我们要用 yepnope 来并行加载多个脚本并按给定次序运行它们。举个例子，假设我们想加载 Backbone.js，而这个脚本又依赖于 Underscore.js。为此，我们只需用数组形式提供这两个脚本

① 参见 http://yepnopejs.com/。

的位置作为加载参数。

```
yepnope({
  load: ['underscore.js', 'backbone.js'],
  complete: function() {
    // 这里是 Backbone 的业务逻辑
  }
});
```

请注意，这里使用了 `complete`（完成）而不是 `callback`（回调）。其差别在于，脚本加载列表中的每个资源均会运行 `callback`，而只有当所有脚本都加载完成后才会运行 `complete`。

yepnope 的标志性特征是*条件加载*。给定 `test` 参数，yepnope 会根据该参数值是否为真而加载不同的资源。举个例子，假设已经使用了 Modernizr 库，则可以以一定的准确度判断用户是否在用触摸屏设备，从而据此相应地加载不同的样式表及脚本。

```
yepnope({
  test: Modernizr.touch,
  yep: ['touchStyles.css', 'touchApplication.js'],
  nope: ['mouseStyles.css', 'mouseApplication.js'],
  complete: function() {
    // 不管是哪一种情况，应用程序均已就绪！
  }
});
```

我们只用寥寥几行代码就搭好了舞台，可以基于用户的接入设备而给他们完全不同的使用体验。当然，不是所有的条件加载都需要备齐 `yep`（是）和 `nope`（否）这两种测试结果。yepnope 最常见的用法之一就是加载垫片脚本以弥补老式浏览器缺失的功能。

```
yepnope({
  test: window.json,
```

```
  nope: ['json2.js'],
  complete: function() {
    // 现在可以放心地用 JSON 了
  }
});
```

页面使用了 yepnope 之后应该变成下面这种漂亮的标记结构：

```
<html>
<head>
  <!-- metadata and stylesheets go here -->
  <script src="headScripts.js"></scripts>
  <script src="deferredScripts.js" defer></script>
</head>
<body>
  <!-- content goes here -->
</body>
</html>
```

很眼熟？这个结构和讨论 `defer` 属性那一节给出的结构一样，唯一的区别是这里的某个脚本文件已经拼接了 yepnope.js（很可能就在 deferredScripts.js 的顶部），这样就可以独立地加载那些根据条件再加载的脚本（因为浏览器需要垫片脚本）和那些想要动态加载的脚本（以便回应用户的动作）。结果将是一个更小巧的 deferredScripts.js。

笔者很喜欢 yepnope。对那些比较简单的应用来说，譬如只想抓取某些垫片脚本的应用，或者只想在用户点击什么东西时再加载某个特性的应用，yepnope 真是做得太完美了。不过，那些非常庞杂的应用仍然需要更强大的技术。

6.3.3　Require.js/AMD的智能加载

开发人员想通过脚本加载器让混乱不堪的富脚本应用变得更规整有

序一些，而 Require.js 就是这样一种选择。Require.js 这个强大的工具包能够自动和 AMD 技术一起捋顺哪怕最复杂的脚本依赖图。

我们稍后再讨论 AMD，现在先来看一个用到 Require.js 同名函数的简单脚本加载示例。

```
require(['moment'], function(moment) {
  console.log(moment().format('dddd')); // 星期几
});
```

`require` 函数接受一个由模块名称构成的数组，然后并行地加载所有这些脚本模块。与 yepnope 不同，Require.js 不会保证按顺序运行目标脚本，只是保证它们的运行次序能满足各自的依赖性要求，但前提是这些脚本的定义遵守了 AMD（Asynchronous Module Definition，异步模块定义）规范。

AMD 规范[①]的宗旨是替浏览器做 CommonJS 标准已经替服务器做过的那些事。（Node.js 模块即基于 CommonJS 标准。）AMD 推行一个由 Require.js 负责提供的名叫 `define` 的全局函数，该函数有 3 个参数：一个模块名称，一个模块依赖性列表，以及一个在那些依赖性模块加载结束时触发的回调。例如，下面的 `define` 语句可以作为依赖 jQuery 之应用模块的有效 AMD 定义。

```
define('myApplication' ['jquery'], function($) {
  $('<body>').append('<p>Hello, async world!</p>');
});
```

注意这里传递给回调的是 jQuery 对象`$`。实际上，`define` 接受的这个回调参数一直对应着依赖性列表中的各个模块依赖项。大家也

① 参见 https://github.com/amdjs/amdjs-api/wiki/AMD。

许奇怪：`define` 怎样知道捕获 jQuery 对象呢？答案是 jQuery 自己的 AMD 定义[①]通过其 `define` 回调返回了 jQuery 对象，借此声明“这是我的导出对象”。

```
define( "jquery", [], function () { return jQuery; } );
```

AMD 的奥妙不止于此，但这是精华所在。如果应用的每个脚本都添加了 AMD 定义，则意味着我们只要调用 `require` 就能保证其回调不被调用，除非既满足了应用对脚本的直接依赖性，又满足了脚本的依赖性和脚本所依赖的那些脚本的依赖性，并且所有脚本均按最大的并行性进行加载，而运行次序也和依赖图一致。

听起来很棒，是吧？但也有一点美中不足：虽然 AMD 已经在 JavaScript 社区产生了一些影响，但仍然有大量的观望者。譬如，Jeremy Ashkenas 就拒绝为其广受欢迎的 Underscore.js/Backbone.js 库添加必要的 AMD 规范，他还在等待着一个 ECMAScript 模块标准。因此，我们不能指望第三方模块都带有自己的 AMD 定义。选择 AMD 会让应用更具一致性，但这也会滋生呆板木讷的代码。

本节介绍了如何通过操控 DOM 来实现运行时加载脚本，还讨论了两个旨在简化脚本加载过程的库：yepone 是一个小巧而精干的工具，Require.js 则是一个巨硕而强大的工具。最终选择哪个库完全取决于正在开发的应用类型和开发团队的类型。“企业级”应用和较大型的前端开发团队更可能受益于 Require.js 大力推行的 AMD 模块化技术。

---

① 参见 https://github.com/jquery/jquery/blob/master/src/exports.js#L17。

6.4 小结

要想让自己的站点更快捷，可以异步加载那些暂时用不到的脚本。为此最简单的做法是审慎地使用 `defer` 属性和 `async` 属性。如果要求根据条件来加载脚本，请考虑像 yepnope 这样的脚本加载器。如果站点存在大量相互依赖的脚本，请考虑 Require.js。选择最适合任务的工具，然后使用它，享受它带来的便捷。

本书内容到此就要结束了，感谢大家花时间阅读。能执笔写这本书是笔者的荣幸。无论大家的 JavaScript 之旅从这里走向何方，笔者都真诚地希望那里会盛开着美丽的、事件驱动式的代码之花。

附录

JavaScript 编辑工具

本附录简要介绍了一些最流行的、套用同步编码风格来编写异步 JavaScript 代码的工具。这些工具都不能替代我们自己对 JavaScript 事件的正确理解，但能为我们的异步智库添加点小插曲。

A.1 TameJS

TameJS 是 OkCupid 团队开发的预编译器项目（http://tamejs.org/），它在 GitHub 上获得了 600 多人的关注。它向 JavaScript 添加了两个关键字：`await` 和 `defer`。`await` 语句块定义的代码只有等到 `defer` 定义的各项异步任务均完成后才能返回。

```
await {
  setTimeout(defer(), 100);
}
console.log("this will run after the 100ms timeout");
```

TameJS 的项目人员还使用这种 `await/defer` 机制基于 CoffeeScript 派生出了 IceCoffeeScript 项目（http://maxtaco.github.com/coffee-script/）。

A.2　StratifiedJS

StratifiedJS（http://onilabs.com/stratifiedjs）是 await/defer 范型的替代品，它提供了粒度更细的一些控制结构，其名称也更直观一些，如 waitfor、resume 等。

```
console.log('this code will run right away...');
waitfor() {
  setTimeout(resume, 500);
}
console.log('...and this will run 500ms later.');
```

尽管 StratifiedJS 未能广泛应用，但这个“野心勃勃”的项目还是受到了 John Resig（jQuery 之父）等人的赞誉。[①]请注意，StratifiedJS 是规范的名称而不是实现的名称，官方的参考实现名称叫做 Oni Apollo[②]。

A.3　Kaffeine

Kaffeine 这个预编译器标榜自己是“扩展版的 JavaScript”。[③]除了其他特性之外，Kaffeine 语言针对非嵌套回调提供了一颗简单的语法糖：只要在异步函数名之后添加一个!。Kaffeine 认为该函数的末参数是一个回调，并简单地把位于函数调用之后的所有代码转换成这个回调。

例如，jQuery 应用经常希望在文档就绪之后在$回调内部运行各种东西。用 Kaffeine 来写就很简单了。

① 参见 https://twitter.com/jeresig/statuses/164496725254479872。
② 参见 https://github.com/onilabs/apollo。
③ 参见 http://weepy.github.com/kaffeine/index.html。

```
$!()
alert('The document is ready.');
```

虽然 Kaffeine 不像 CoffeeScript 那样拥有一大批支持者，但类似这样的特性也值得我们关注它。基于 CoffeeScript 派生出来的 CoCo 和 LiveScript 这两个项目也有相似的特性，称作“*后置调用*”。用 CoCo/LiveScript 改写上述示例后形如：

```
<- $
alert 'The document is ready'
```

Kaffeine、CoCo、LiveScript 等项目提供的功能和 TameJS/StratifiedJS 差不多，但前者更易学习。

A.4　Streamline.js

Streamline.js[①]类似于 Kaffeine，也提供了一种特殊语法，负责将异步函数调用之后的代码转换成它的回调。只要在回调参数的位置写上_即可。

```
for (var s = 1; s < 60; s++) {
  setTimeout(_, 1000);
  console.log(s + ' seconds have elapsed');
}
console.log('1 minute has elapsed');
```

有趣之处在于，Streamline 既可以生成标准的回调驱动式 JavaScript，又可以生成用于 node-fibers 的基于纤程的代码（参见 A.5）。此外，Streamline 与大多数其他的 JavaScript 预编译器不同，它可以与 CoffeeScript 联合使用。

① 参见 https://github.com/Sage/streamlinejs。

注意，Streamline 希望回调能遵守 Node 风格的参数列表约定(`err, results...`)，这虽然简化了 Node 环境下的错误处理，却很难用在浏览器开发环境中。

A.5 Node-Fibers

附录里罗列的其他项目都只是编译成平平常常的 JavaScript，但 node-fibers 事实上已经通过 Node 运行时扩展了对 JavaScript 的理解，它添加了一种类似于线程的构造——纤程（fiber，http://en.wikipedia.org/wiki/Fiber_(computer_science)）。纤程可以让步于其他纤程，可以暂停自己的执行逻辑，等到某个事件导致自己再运行。

```
var fiber = Fiber.current;
console.log('Yielding until the timeout elapses...')
setTimeout(function() {
  fiber.run();
}, 1000);
Fiber.yield();
console.log('...1 second later');
```

相对于那些 JavaScript 预编译器，node-fibers 的主要优势在于调试。node-fibers 模块在堆栈轨迹中的行号对应于源代码中的行号，所抛出的异常即使当纤程处于 `try/catch` 语句块内部时也可以被捕获。

A.6 JavaScript的未来：生成器

最新版的 ECMAScript（所有主流 JavaScript 运行时环境均需实现的规范）定义了一种新的 JavaScript 特征——生成器（generator）。生成器是一种特殊的、含有 yield 语句的函数类型。`yield` 类似于 `return`，

只不过生成器下次再运行时会从这条 `yield` 语句重新开始。

从概念上看，生成器有点复杂。不过 Mozilla 的 Task.js（http://taskjs.org/）库展示出这种生成器如何让异步代码更简单。由于生成器可以重新开始，我们能写出类似这样的代码：

```
task.spawn(function() {
  console.log("Yielding...");
  yield task.sleep(1000);
  console.log("...resuming 1 second later");
});
```

下面简述这段代码的工作方式。`task.spawn` 负责运行生成器函数。如果没有 `yield`，生成器函数不过是个普通的函数。`task.sleep` 负责返回一个 1000 毫秒后即执行的 Promise 对象（请参见第 3 章）。`task.spawn` 接受这个 Promise 对象,并附加生成器作为 Promise 对象执行时的回调。Promise 对象执行之后，会再次调用生成器，这时会从 `yield` 语句后继续执行。听起来有点复杂，不过活儿干得确实很漂亮。

ECMAScript 6（项目代号 Harmony）迄今仍未有定论。目前，生成器唯一的重要实现出现于 Firefox 浏览器，而且只有当`<script>`标签中定义 JavaScript 版本至少为 1.7 时才能启用这一特性，形如：

```
<script type = "application/javascript; version=1.7">
```

如果生成器深深打动了你，有个工具允许把使用生成器（包括其他 ECMAScript 5+特性）的代码编译成广泛支持的 JavaScript 代码，即 Google 的 Traceur 项目[①]。

① 参见 http://code.google.com/p/traceur-compiler/。

索　　引